conference

proceedings

7th USENIX Tcl/Tk Conference

Austin, TX, USA
February 14–18, 2000

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $24 for members and $32 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-24-3

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the 7th USENIX Tcl/Tk Conference (Tcl/2k)

February 14–18, 2000
Austin, Texas, USA

Conference Organizers

Contents

7th USENIX Tcl/Tk Conference

February 14–18, 2000
Austin, Texas, USA

Message from the Conference Chairs v

Wednesday, February 16

Middleware
Session Chair: Melissa Chawla, Scriptics Corp.

Rapid CORBA Server Development in Tcl: A Case Study 1
Jason Brazile and Andrej Vckovski, Netcetera AG

AGNI: A Multi-Threaded Middleware for Distributed Scripting 11
M. Ranganathan, Marc Bednarek, Fernand Pors, and Doug Montgomery, National Institute of Standards and Technology

Introducing QoS Awareness in Tcl Programming: QTcl 23
Roberto Canonico, Maurizio D'Arienzo, Simon Pietro Romano, and Giorgio Ventre, Università di Napoli "Federico II"

CollabWiseTk: A Toolkit for Rendering Stand-alone Applications Collaborative 31
Hemang Lavana and Franc Brglez, North Carolina State University

Testing and Integration
Session Chair: Dave Griffin, SiteScape, Inc.

GDBTk: Integrating Tcl/Tk into a Recalcitrant Command-Line Application 43
Jim Ingham, Cygnus Solutions

TclTk: A Strong Basis for Complex Load Testing Systems 53
Ahmet Can Keskin, Till Immanuel Patzschke, and Ernst von Voigt, Patzschke + Rasp Software AG

Using Tcl to Build a Buzzword-Compliant Environment That Glues Together Legacy Analysis Programs 61
Carsten H. Lawrenz and Rajkumar C. Madhuram, Siemens Westinghouse Power Corp.

Thursday, February 17

Web Technologies
Session Chair: Cameron Laird, Phaseit, Inc.

Proxy Tk: A Java Applet User Interface Toolkit for Tcl 71
Mark Roseman, TeamWave Software, Ltd.

The TclHttpd Web Server 81
Brent Welch, Scriptics Corp.

TkGecko: A Frill-Necked Lizard 97
Steve Ball, Zveno Pty Ltd.

Scriptics Connect 105
Eric Melski, Scott Stanton, and John Ousterhout, Scriptics Corp.

User Interface and Applications
Session Chair: Dave Beazley, University of Chicago

Supporting Information Awareness Using Animated Widgets 117
D. Scott McCrickard and Q. Alex Zhao, Georgia Institute of Technology

Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience 129
Franc Brglez, Hemang Lavana, Zhi Fu, Debabrata Ghosh, Lorie I. Moffitt, Steve Nelson, J. Marshall Smith, and Jun Zhou, North Carolina State University

Scripted Documents 141
Jean-Claude Wippler, Equi4 Software

Friday, February 18

Extending Core Tcl
Session Chair: Matt Newman, Sensus Consulting Ltd.

The Tcl Extension Architecture 151
Brent Welch and Michael Thomas, Scriptics Corp.

XOTcl—An Object-Oriented Scripting Language 163
Gustaf Neumann, Vienna University of Economics and BA, and Uwe Zdun, University of Essen

A Multi-Threaded Server for Shared Hash Table Access 175
Andrej Vckovski and Jason Brazile, Netcetera AG

Feather: Teaching Tcl Objects to Fly 183
Paul Duffin, IBM Hursley Laboratories, UK

Message from the Conference Chairs

Welcome to the first Tcl/Tk conference of the new millennium!

Once again, the good folks at USENIX and the Tcl community have combined forces to bring you an interesting and involving program of tutorials, papers, posters, BoF sessions, and informal activities. Tcl/Tk has gained significant visibility and "mind share" over the last couple of years; during our refereed paper presentations you'll see and hear how the language is being applied to current, practical problems in Web-based technology, distributed applications, and more.

This year's keynote speaker is Jim Davidson, VP of Technology at AOL's Digital City. He'll tell you about the essential role that Tcl plays in AOLserver, and why Tcl was chosen for this highly visible application.

Our tutorial program this year is threaded for both beginners and expert Tcl programmers. Topics range from the proper use of the mysterious and powerful regexp command, to the internationalization of Tcl applications—in addition to perennial favourites like introductions to [incr Tcl] and Expect. The tutorial program is always popular at Tcl/Tk conferences; we hope you'll find it not only useful but fun as well.

Although the conference chairs' names appear at the top of the program, lending us the illusion of importance, in reality the conference is always the creation of many people working as a team. We'd like to offer our sincere thanks to the ever-patient, helpful, and diligent USENIX staff, to our program committee for their thoughtful reviews of submitted papers, and to our author/presenters for the work they put into their papers and posters. We hope you will find this first 21st-century Tcl conference both enjoyable and informative, and we look forward to seeing you again at future conferences.

De Clarke, *UCO Lick Observatory*

Tom Poindexter, *Talus Technologies, Inc.*

Rapid CORBA Server Development in Tcl: A Case Study

Jason Brazile and Andrej Vckovski
Netcetera AG
{jason.brazile,andrej.vckovski}@netcetera.ch

Abstract

A large Swiss bank needed to collect, combine, process, and distribute financial market data from various 3rd party data sources to a large number of internal and external clients – the typical integration task at which scripting languages excel. The bank uses an implementation of CORBA as their standard enterprise-wide middleware for distributed applications. We describe how we designed and built a Tcl/C++ transport framework which allowed us to develop the "kernel" of this server application entirely in Tcl, yet support CORBA as the primary interface to the server. We further describe how this framework allows a small development team to rapidly implement changes and enhancements to the server and its external interface, while automatically generating the corresponding changes that are needed for the CORBA interface. Additionally, we show how we were able to automatically generate code to create new tcl commands that make use of the same, generated, Tcl/C++ marshalling routines in order to develop a CORBA client in Tcl, which is used to regression test the server, when full end-to-end testing is needed.

1 Overview

In his keynote address at the 1999 USENIX Technical Conference [1], John Ousterhout argued that typical programming projects are shifting away from large stand-alone applications and moving toward *integration* applications, (sometimes also referred to as *mega-programming* [2]). The importance of these applications, he argued, lies not in providing fundamental new features as much as in the ability to coordinate and extend existing applications – exactly the tasks at which scripting languages excel. He argued that many things taken for granted today, such as strong typing and inheritance, may not make sense in most future applications.

As we analyzed our customers requirements for for their strategic new mission critical server application, we found ourselves in agreement with this basic philosophy. It became clear that our customer needed to primarily integrate information coming in from several already existing applications – combining, merging, and formatting everything into a unified view. This result would need to be exported through certain transport mechanisms – initially the most important being CORBA. However, even at the earliest stages, other access channels (e.g. XML) were planned.

Upon noting CORBA as a requirement, many software architects might automatically dismiss the idea of designing the core application functionality in a scripting language such as Tcl, even though it is not without precedent [3], [4]. However, we were further encouraged to follow our core-as-scripting-language idea when we realized that our application needed to be prepared for the following:

- Frequent changes to the interfaces to external applications
- Rapid integration of new data sources (i.e., new applications)
- Frequent changes to the access requirements requested by clients (i.e., IDL changes)
- Rapid implementation of new features requested by marketing analysts
- Possible implementation of different data access channels (e.g. XML) as some move out of favor, and others move in

It is worth mentioning that this server application is mission critical in the sense that some client applications planned to present the data on the Internet (i.e., they are highly visible) for tasks such as financial planning, online banking, and as a financial news source. An even

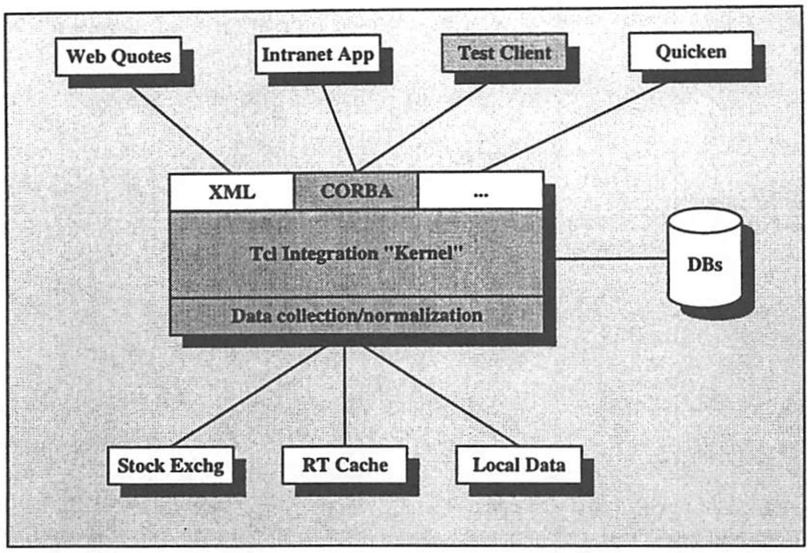


Figure 1: Simplified Server Overview

greater sense of urgency was placed upon us when we learned that some of the client applications were planned to go online within weeks of our planned initial release – many of these applications with budgets 10 times larger than ours.

However, these robustness and flexibility requirements merely further increased our resolve to attempt to develop as much as possible in a scripting language and as little as possible in traditional CORBA server implementation languages.

Our goal then became to design and implement a flexible platform that performed these integration tasks – providing data in a canonical form and otherwise acting as mediator to a large and growing set of heterogeneous data sources with different manners of access to the same interfaces.

2 Architecture

The overall feature that we were targetting was a one-to-one mapping between the methods defined by the CORBA IDL (Interface Definition Language) specification and the Tcl procedures that would implement these specifications. To illustrate this with an example, consider the following IDL definition for a method called `foo::square()`:

```
interface foo {
  int square (
    in  int a,
    out int b
    );
}
```

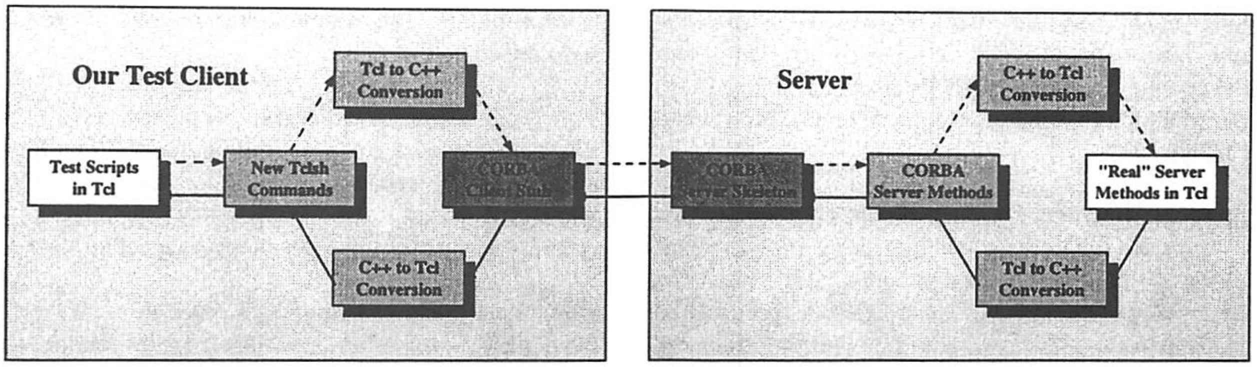
We would like to define a Tcl procedure to implement this method that looks like this:

```
proc foo::square {a vb} {
   upvar $vb b
   set b [expr $a * $a]
}
```

The general idea would be that a CORBA request coming in to the server would get translated to a corresponding Tcl command. Then a Tcl interpreter would be invoked that first "sourced" the Tcl implementations of these commands, and then "eval'd" the Tcl command that was composed. It would then translate the Tcl results of that evaluation back into CORBA objects which are returned to the client.

With this design, the framework would then consist of a CORBA server with an embedded Tcl interpreter that would source the "real methods" implemented as procedures in a Tcl script "kernel".

However, in addition, we wanted to extend a Tcl shell that would serve as a CORBA client which would con-



■ **C++ generated by CORBA IDL compiler (~11,200 lines of code)**
▨ **C++ generated by Tcl scripts based on IDL definition (~10,600 lines of code)**
☐ **Tcl written "by hand" (~11,800 lines of code)**

Figure 2: Data Flow (CORBA Case)

tain Tcl commands that also had a one-to-one mapping to the methods defined in the IDL. The function signatures of these new commands would look identical to the function signatures in the procedures in the Tcl "kernel". This would allow for natural looking Tcl code as in the following example:

```
% foo::square 4 result
% set result
16
```

One of the best characteristics of this architecture is that test scripts could be written in Tcl that could be run either on the CORBA Tcl client or in a "shortcut" path which completely bypasses any middleware infrastructure by making direct access to the Tcl "kernel" procedures. This feature allowed much of the server functionality to be able to be developed on a stand-alone UNIX laptop which did not have a CORBA development environment.

To summarize, this design approach would give us the following benefits:

- The ability to begin development of the core methods in Tcl completely independent of, and in parallel to, the development of CORBA support.
- The ability to write test scripts in Tcl
- The ability to develop all the core programming logic in a rapid turnaround scripting language, rather than a lower level, strictly typed, compiled language.
- If different transport paths are required, only a new transport layer would need to be written and the same "kernel" could be used to give identical functionality with maximal code reuse.

After implementing a small prototype to test the feasibility of this approach, it became clear that a large amount of the CORBA transport layer code was regular enough that it could be automatically generated – again by scripts written in Tcl.

The full data path and amount of automatic code generation we ended up with is shown in Figure 1.

At the end of the design phase, we concluded that the key aspects of this architecture required the following components to be written:

- CORBA Language mappings for Tcl
- Methods/Operations implemented in Tcl
- Automatic generation of the marshalling code based on IDL definitions
- Automatic generation of client side tcl commands based on IDL definitions

3 Language mappings

One of the key issues to resolve was how to map CORBA objects to typeless Tcl strings. In the worst case, this could be done by defining every C++ value to be a Tcl list containing two elements – the value and its type. Then, a lookup table could be constructed to store the types and whatever needed semantics might be associated with it in order to manipulate or access objects of that type.

Fortunately, however, a much simpler strategy was possible in this case. We ended up using a mapping similar to that used by Pilhofer in Tclmico [3]. We started by looking at all of the possible CORBA types that are made available – the basic data types like boolean, integer, and floats, as well as the compound data types such as structures, arrays, unions, and sequences. We also wanted to be able to map Tcl exceptions to CORBA exceptions.

It is fairly straightforward to map the common types such as a structure. For example, an object of type `RequestContext` which can be defined like this:

```
typedef sequence <string> Profile;
struct rc {
  int      sessionId,
  string   application,
  string   lang,
  Profile  profile,
  string   user,
};
typedef rc RequestContext;
```

could become a Tcl list with the following representation:

```
set my_rc {-1 web-quotes ENGLISH \
    {SWISS_REALTIME US_DELAYED} guest}
```

Going from this Tcl list representation back to its corresponding C++ object representation requires that we know the CORBA types of each component. However, this is positionally implicit based on the context of the procedure calls that use variables of these types and the definitions of the procedures themselves as defined in the IDL. In other words, when we know that something is the first argument to method `getMarketData` and according to our codification of the IDL, the first argument to method `getMarketData` is of type `RequestContext`, then we just pass this list to a routine that converts a list (which in this case, itself contains a list) to a C++ object of type `RequestContext`. This can be done statically, because we always know at compile time which procedures are being called and what types their arguments are because of their position. That is, except in the case of translating "exceptions", which is described below.

To take a quick glance at some of the other types, a variable whose type is a `union` can be mapped to a two-element list where the first element is the discriminator and the second is the corresponding value. The `sequence` and `enum` types are simply mapped to a list.

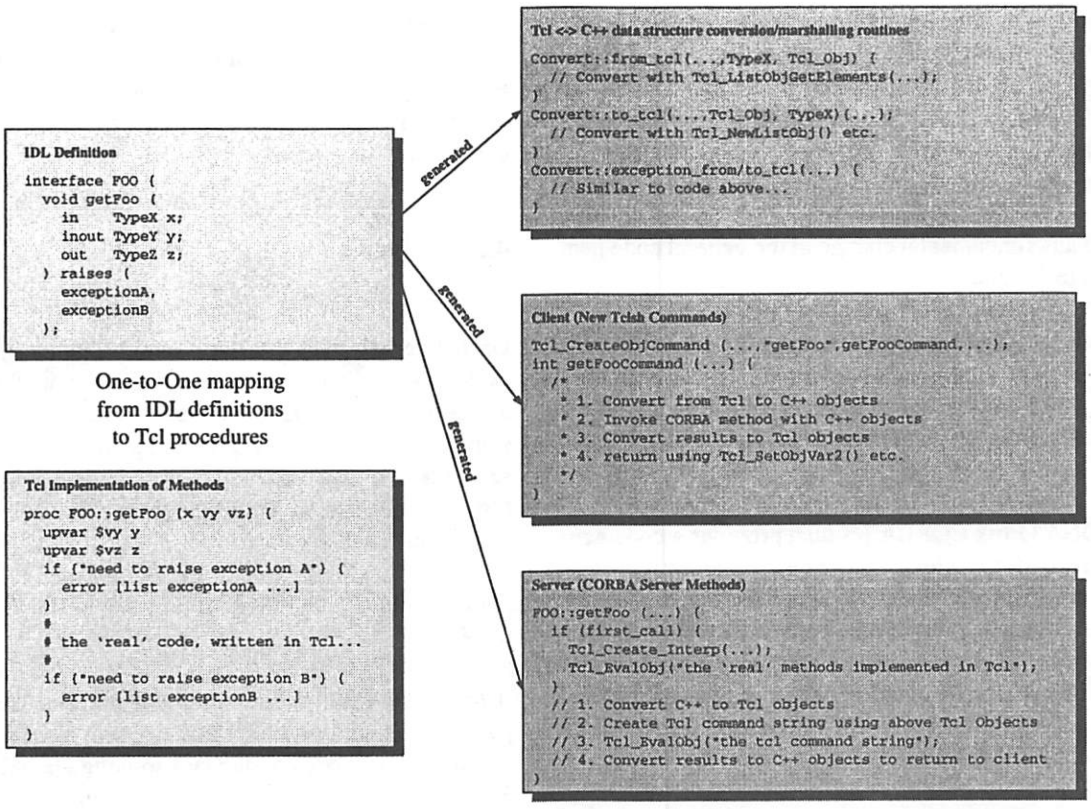
Perhaps the trickiest mapping to come up with was a mapping for exceptions. A Tcl exception (as thrown by the `error` command) allows only one string/list as an argument. However, for our purposes, we needed to know both *what* exception occurred (i.e., an exception type) and an additional exception-specific object. The problem of course is that this second object's "shape" can be different depending on what type of exception is to be thrown.

One interesting point to be made is that by having a general mapping mechanism for Tcl exceptions, our application can catch not only application specific exceptions that we throw ourselves, but also the standard Tcl exceptions that may occur due to attempted undefined variable access, for example.

We imposed a convention upon our own code whereby our application specific exception objects would have a certain well-defined structure, so that we could then differentiate them from Tcl exceptions which get mapped to an "Internal Error" exception with a Tcl backtrace, rather than merely crashing the application.

This turned out to be the one case where we needed to determine type information dynamically in order to be able to do the CORBA language mappings. The code is structured as follows:

```
if (Tcl_EvalObj() == TCL_ERROR) {
  if (/* has special shape */) {
    type = Tcl_GetStringFromObj();
    ex_type = type_to_enum(type);
    switch (ex_type) {
    case exceptionA:
      // now we know the type
      Convert::from_tcl(...);
      THROW(exceptionA, ...);
    case exceptionB:
      // now we know the type
      Convert::from_tcl(...);
      THROW(exceptionB, ...);
    }
  } else {
    // Tcl Exception
    THROW(internal, ...);
```



IDL Definition

```
interface FOO {
  void getFoo (
    in    TypeX x;
    inout TypeY y;
    out   TypeZ z;
  ) raises (
    exceptionA,
    exceptionB
  );
```

Tcl Implementation of Methods

```
proc FOO::getFoo {x vy vz} {
  upvar $vy y
  upvar $vz z
  if {"need to raise exception A"} {
    error [list exceptionA ...]
  }
  #
  # the 'real' code, written in Tcl...
  #
  if {"need to raise exception B"} {
    error [list exceptionB ...]
  }
}
```

Tcl <-> C++ data structure conversion/marshalling routines

```
Convert::from_tcl(...,TypeX, Tcl_Obj) {
  // Convert with Tcl_ListObjGetElements(...);
}
Convert::to_tcl(...,Tcl_Obj, TypeX)(...);
  // Convert with Tcl_NewListObj() etc.
}
Convert::exception_from/to_tcl(...) {
  // Similar to code above...
}
```

Client (New Tclsh Commands)

```
Tcl_CreateObjCommand (...,"getFoo",getFooCommand,...);
int getFooCommand (...) {
  /*
   * 1. Convert from Tcl to C++ objects
   * 2. Invoke CORBA method with C++ objects
   * 3. Convert results to Tcl objects
   * 4. return using Tcl_SetObjVar2() etc.
   */
}
```

Server (CORBA Server Methods)

```
FOO::getFoo (...) {
  if (first_call) {
    Tcl_Create_Interp(...);
    Tcl_EvalObj("the 'real' methods implemented in Tcl");
  }
  // 1. Convert C++ to Tcl objects
  // 2. Create Tcl command string using above Tcl Objects
  // 3. Tcl_EvalObj("the tcl command string");
  // 4. Convert results to C++ objects to return to client
}
```

Figure 3: Automatic Generation of CORBA "Transport" Routines

```
    }
  }
```

Raising an exception from within our Tcl code looks like this:

```
  if {$some_exceptional_condition} {
    error {badParam \
      {431 "FOO parameter expected"}}
  }
```

4 Code Generation

One of the most important features of the framework is that a large portion of the low-level (e.g. non-Tcl) code can be automatically generated, (Figure 2). This is good not only for quick response to changes in requirements, but also in our level of confidence in the server's robustness. Once we developed a logical, regular pattern for use in code generation, and we measure and test that code for relative correctness and memory leaks etc., then we have a high degree of trust in code that is generated from the same patterns in future enhancements. And as a practical matter, we feel confident that if the generated code checks return values and correctly handles error conditions in one case, it will also correctly handle them in all similar cases.

To recap, our design revealed that the following components could be generated automatically:

- Marshalling code from/to Tcl
- Implementation of new Tcl commands for a Tcl interpreter used as a CORBA client
- CORBA server skeletons which translate a method to a Tcl command, "eval" the Tcl "kernel", and translate the result back.

We used our hand-written prototype implementation as a model for the generated C/C++ code. There is nothing particularly noteworthy about the Tcl scripts which generated the C/C++ code. It was mostly a matter of template textual substitution.

It should be pointed out that the code generation scripts are not necessarily general purpose. Code was only written to generate the types and cases that appeared in our server IDL definition. As new methods and data types were added to the IDL definition, additional lines of code generation code occasionally had to be written. On the other hand, none of these additions so far have caused any fundamental change in the general code generation technique that is used.

It should further be pointed out that the code is not generated *directly* from the IDL definition files, but rather from a "pre-parsed" intermediate representation of the information that is represented in the IDL definition files.

The IONA Orbix CORBA product provides a code generation toolkit [5] which exposes a pre-parsed representation of IDL files through Tcl data structures and accessor procedures by way of a built-in Tcl interpreter. This would be an efficacious approach for our application, but at the time of implementation, we were limited to a proprietary CORBA environment which therefore required our own ad hoc solution.

Our simplistic intermediate file representation was designed so that it can be used in a Tcl code generation script by merely "eval'ing" it. We wanted to defer the difficult problem of parsing the IDL files themselves in the hopes that an automatic solution, such as that provided by Orbix, would eventually become available. In the meantime, making these changes to our intermediate file "by hand", whenever the IDL file has changed, has not required a significant effort.

4.1 Marshalling

The marshalling code was built around the idea of having two essential polymorphic conversion methods – `Convert::from_tcl()` which would take a Tcl object representing a list structure as input and produce a CORBA object (which may be composed of CORBA sub-objects) as output – and `Convert::to_tcl()` which would go the opposite direction. The types of the arguments to these routines would determine which instance of the conversion routine would be called.

It was also decided that these routines would recursively call themselves to process each subcomponent that might need to be converted, thus eliminating code duplication.

A representative pair of automatically generated conversion methods is shown for our `RequestContext` object in Figure 3. Notice that the fourth member of the structure is a non-primitive object which leads to a recursive call to `Convert::from_tcl`.

4.2 Server

On the server side, the IDL compiler generates server skeletons (i.e. procedure stubs) for the methods that are defined in the IDL. However, our Tcl script subsumes this behavior by generating its own completed server skeletons which merely rely on there being a correspondingly named Tcl procedure in the Tcl "kernel" that it can "eval".

An example of generated server code is shown in Figure 4.

It might be worth mentioning that the Tcl "kernel" is packaged as a monolithic static string in a shared object so that it appears as a standard looking shared library. This is advantageous not only for simpler code distribution, but has a higher management acceptance factor than scripts as text files.

4.3 Client

On the client side, much of the code is analogous to what is required for the server. The arguments need to be converted, the relevant method in the new implementation language needs to be called, then the output from the result needs to be translated back.

An example of generated client code is shown in Figure 5.

5 Testing

By far, the greatest advantage in testing was the ability to write tests in Tcl and to merely source our "kernel" implementations of the methods being tested. A quick change could be made and the method in question could be re-"sourced" with rapid turnaround.

Also, with this method, the important code could be written without the need for a CORBA development in-

```
int Convert::from_tcl(Tcl_Interp *interp,
      RequestContext *&requestContext,
      Tcl_Obj *tcl_obj)
{
  Tcl_ListObjGetElements(interp, tcl_obj, &obj);
  /* verify that there are 5 arguments */
  Tcl_GetLongFromObj(..., obj[0], &(requestContext->sessionId));
  requestContext->application = Tcl_GetStringFromObj(..., obj[1], ...);
  requestContext->lang = Tcl_GetStringFromObj(..., obj[2], ...);
  Convert::from_tcl(interp, &(requestContext->profile), obj[3]);
  requestContext->user = Tcl_GetStringFromObj(..., obj[4],...);
}
int Convert::to_tcl(Tcl_Interp *interp,
      Tcl_Obj **tcl_obj,
      RequestContext *requestContext)
{
  *tcl_obj = Tcl_NewListObj(0, NULL);
  Tcl_ListObjAppendElement(Tcl_NewLongObj(requestContext->sessionId,...));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->application,...));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->lang,...));
  Convert::to_tcl(interp, &tcl_profile, &(requestContext->profile));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->user,...));
}
```

Figure 4: Simplified Data Marshalling code (CORBA Case)

```
void MDS::getMarketData (
  /* in */  const RequestContext &requestContext,
  /* in */  const DataSelector   &dataSelector,
  /* in */  const ReturnFields   &returnFields,
  /* out */ const MarketData     &results,
)
{
  interp = Tcl_CreateInterp();
  Tcl_EvalObj(/* the "kernel" */);
  Convert::to_tcl(interp, &tcl_request_context_obj, &requestContext);
  Convert::to_tcl(interp, &tcl_data_selector_obj, &dataSelector);
  Convert::to_tcl(interp, &tcl_return_fields_obj, &returnFields);
  tcl_command_obj = Tcl_NewListObj(0, NULL);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, /* "getMarketData" */);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_request_context_obj);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_data_selector_obj);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_return_fields_obj);
  Tcl_EvalObj(interp, tcl_command_obj);
  tcl_market_data_results_obj =
    TclObjGetVar2(interp, /* "results" */, ...);
  Convert::from_tcl(interp, &results, &tcl_marketdata_results);
}
```

Figure 5: Simplified Generated Server code (CORBA Case)

```
void getMarketDataObjCmd(Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[])
{
  orb_init(&corba_obj);
  Convert::from_tcl(interp, &objv[0], &requestContext);
  Convert::from_tcl(interp, &objv[1], &dataSelector);
  Convert::from_tcl(interp, &objv[2], &returnFields);
  TRY (
    corba_obj->getMarketData(requestContext, dataSelector,
      returnFields, marketDataResults);
  )
  CATCH (InternalError) { /* Handle Exception */ }
  CATCH (badRequestContext) {/* Handle Exception */ }
  Convert::to_tcl(interp, &tcl_marketdata_results, MarketdataResults);
  Tcl_ObjSetVar2(interp, /* "marketDataResults" */, tcl_marketdata_results);
}
```

Figure 6: Simplified Generated Client code (CORBA Case)

frastructure or even the need to run separate server/client processes.

Once we had a CORBA Tcl client, we were able to reuse the same testing scripts we wrote during development to do end-to-end testing across the CORBA channel. This even allowed other development groups who were writing CORBA-based client applications to our server, to use our CORBA Tcl client in order to do quick exploration and cross-checking of the interface whenever they ran into difficulties.

An example test script looks like this:

```
% set requestContext {$sessionID \
    $applicationID ENGLISH \
    {DELAYED} $userId}
% set dataSelector {BYCONSTRAINTS \
    {{} {"US Dollars"} \
    {"Swiss Exchange"}}}
% set returnFields   {BYFIELDNAME \
    {ID CURRENT_PRICE HIGH_PRICE}}
% getMarketData $requestContext \
    $dataSelector $returnFields \
    results
% puts $results
{ID 324598234 CURRENT_PRICE 45.5
HI_PRICE 48.25} {ID 43098234 ...}
```

6 Conclusion

In summary, with our CORBA-as-transport design, we were able to achieve:

- Rapid development
- Robustness
- Ease of testing
- Rapid implementation of interface changes
- Development outside of huge support environment
- Reuse of useful tools in the OSF arena to integrate other applications and leverage all of this even within a rigidly specified middleware framework.

At least up to a certain point, performance was never a high-priority requirement. However, it turned out not to be a problem either. When performance problems were encountered, the largest gains were achieved by adding caching at the Tcl "kernel" level to compensate for slow data accesses to external applications.

The most unexpected benefit was the ease in providing an XML access method to our Tcl "kernel" which obviously gives identical semantics, results, and performance characteristics. We were able to use many of the same code generation techniques to write automatic XML to Tcl marshalling/unmarshalling code, which relies on exactly the same pre-parsed intermediate representation of the IDL that the CORBA transport layers uses.

We must admit that there was initial skepticism in the approach we took, especially at the beginning of the project when so much effort seemed to be needed in just building framework and code generation tools which didn't lead directly to our tightly scheduled goal.

However, in the end we not only achieved our goal but were easily able to adapt to even more change requests than anticipated, given the high level of flexibility our system afforded. The IDL specification has gone through 12 revisions since the application was first launched in December 1998.

7 References

1. Ousterhout, John, *Integration Applications: The Next Frontier in Programming*, Keynote Address, 1999 USENIX Technical Conference, Monterey, California, 1999.

2. Wiederhold, Gio, Peter Wegner, and Stefano Ceri, *Towards Megaprogramming*, Communications of the ACM, Vol.,35 No.11, November 1992.

3. Miller, Michael and Kareti, Srikumar, *Using Tcl to Script CORBA Interactions in a Distributed System*, The Sixth Annual Tcl/Tk Conference, San Diego, California, 1998.

4. Pilhofer, Frank, *Tclmico – A Tcl interface to the Mico ORB* `<http://www.vsb.informatik.uni-frankfurt.de/~mico>`

5. *Orbix Code Generation Toolkit Programmer's Guide*, IONA Technologies PLC, Dublin, Ireland, February 1999.

AGNI: A Multi-threaded Middleware for Distributed Scripting

M.Ranganathan, Marc Bednarek, Fernand Pors and Doug Montgomery
Internetworking Technologies Group
National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899.
{mranga, bednarek, pors, dougm}@antd.nist.gov

Abstract

We present the design of a AGNI - a Tcl 8.1 based Middleware for building reactive, extensible, reconfigurable distributed systems, based upon an abstraction we call Mobile Streams. Using our system, a distributed, event-driven application can be scripted from a single point of control and dynamically extended and re-configured while it is in execution. Our system is suitable for building a wide variety of applications; for example, distributed test, conferencing and control-oriented applications. We illustrate the use of our system by presenting example applications.

1 Introduction

Our work is motivated by a couple of observations about the evolving nature of distributed applications and their implementation environments. First, the structure of distributed software is undergoing some changes. We are seeing the growth of new types federated, loosely coupled distributed applications that have several common requirements and characteristics including: (1)**Event-driven Architecture:** A single distributed application may be composed of separate components that all work together in a coordinated fashion. Such applications are event-oriented in nature in that we can think of changes in the overall state of the global application as being triggered by discrete changes in the state of event processing at each of the components. (2)**Heterogeneity:** The components must run on a variety of different platforms with varying inherent capabilities and environments. In addition, some of components themselves are reused pieces of software implemented in a variety of languages and environments. Despite all of this heterogeneity, it is desirable to be able to design and develop distributed applications in a common portable framework. (3)**Mobility:** The ability to dynamically move code to and among these platforms during application execution greatly enhances the ability to deploy and reconfigure complex systems. (4)**Reliability:** These extension and reconfiguration capabilities can be used to construct reliable systems that distribute system state in ways that enable graceful failure recovery and adaptation. (5)**Security:** In such highly dynamic systems, the ability to secure and control resources at several levels (global application, single node, individual process) is necessary to insure the correct behavior of applications and the viability of systems that support them.

Our second observation is that while many systems share the characteristics mentioned above, the variety of component types and platforms that must be accommodated preclude a language specific or application specific solutions. Instead we suggest that a Middleware approach based upon Tcl scripting technology provides the most flexibility in the design of such composite applications. Tcl is great "component glue". Its simplified structure and extensibility are strengths when assembling applications from disparate, heterogeneous software components [1].

This paper is about AGNI - a multi-threaded Tcl 8.1 based Middleware for scripting reconfigurable event-oriented distributed systems. AGNI builds upon the proven scripting power of Tcl by adding extensions for an abstraction we call Mobile Streams. Mobile Streams (*MStreams*) are a generalization of simple mobile code technologies (e.g. Java Applets) that provide code distribution and communication between clients and servers. MStreams extend today's simple notions of code mobility by incorporating state mobility, decentralized peer-to-peer communications and the ability to extend and reconfigure distributed application during execution while preserving behavioral guarantees. At a higher level, MStreams allow the separation of the logical structure

[1] Indeed, thus spake John Ousterhout : "If you look at financial services, a lot of what they do is try and tie together all of the different systems that need to be coordinated with traders, not to mention the front and back office. It's a tremendous integration effort, and that's exactly what Tcl does wonderfully...".

of a distributed application from the physical placement of components. Our model of code mobility allows the mapping of logical application structure to physical resources (e.g. machines and processes) to occur dynamically at run time and change during the course of the active life-time of the global application.

The rest of this paper is organized as follows: Section 2 presents the MStreams programming model and system architecture and presents an introductory example. Section 3 gives a brief overview of *AGNI*, our prototype implementation of MStreams Middleware. Section 4 presents a simulation environment for designing applications using our system. Section 5 presents some more comprehensive applications that we have built on our system. In Section 6 we compare and contrast our work with those of others. In Section 7 we conclude and present our future plans for this project.

2 Mobile Streams

In this section we present our programming model and provide a small, introductory example. We begin by introducing a few terms that are used through the rest of the paper.

A *Mobile Stream* (*MStream*) is a mobile communication endpoint in a distributed system. The closest analogy to an MStream is a mobile active mailbox. As in a mailbox, an MStream has a globally unique name. *MStreams* provide a *FIFO* ordering guarantee, ensuring that messages are consumed at the MStream in the same order as they are sent to it. Usually mailboxes are stationary. MStreams, on the other hand, have the ability to move from *Site* to *Site* dynamically. Usually mailboxes are passive. In contrast, message arrival at an MStream potentially triggers the concurrent execution of message consumption event handlers (*Append Handlers*) registered with the MStream, which can process the message and, in turn, send (*append*) messages to other MStreams.

An MStream has a globally unique name. We refer to any processor that supports an MStream execution environment as a *Site*. A distributed system consists of one or more Sites. A collection of Sites participating a distributed application is called a *Session*. Each Session has a distinguished, trusted, reliable Site called a *Session Leader*. Each Site is assigned a *Location Identifier* that uniquely identifies it within a given Session. New Sites may be added and removed from the Session at any time. An MStream may be located on, or moved to any Site in the Session that allows it to reside there. MStreams may be opened like sockets and messages sent (appended) to them. Multiple Event Handlers (*Handlers*) may be dynamically attached, to and detached from, an MStream. Handlers are invoked on discrete changes in system state such as message delivery (append), MStream relocations, new Handler attachments new Site additions and Site failures. We refer to these discrete changes in system state as Events. Handlers are attached by *Agents* which provide an execution environment and thread for the Handlers that they attach. (i.e. an Agent specifies a collection of Handlers that that all use the same thread of execution and interpreter.) Logically, the system is structured as shown in Figure 1.

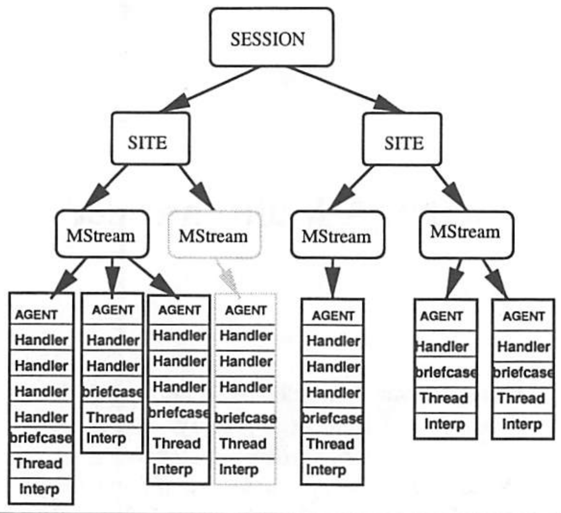
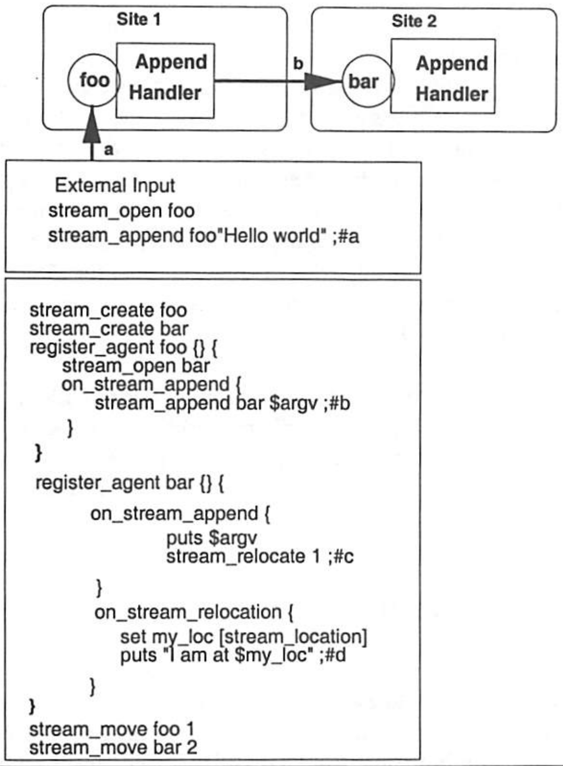


Figure 1: Logical organization of the System. A Session consists of multiple participating Sites. Each Site can house multiple MStreams. Each MStream can have multiple Agents that can register Handlers for different Events. MStreams can move from Site to Site. When an MStream moves, all its registered handlers move with it.

Handlers can communicate with each other by appending messages to MStreams. These messages are delivered asynchronously to the registered Append Handlers in the same order that they were issued [2]. A message is *delivered* at an MStream when the Append Handlers of the MStream has been activated for execution as a result of the message. A message is *consumed* when all the Append Handlers of the MStream that are activated as a result of its delivery have completed execution. By *asynchronous delivery* we mean that the sender does not block until the message has been consumed in order to continue its execution.

[2] Synchronous delivery of messages is supported as an option but asynchronous delivery is expected to be the common case.



```
External Input
stream_open foo
stream_append foo"Hello world" ;#a
```

```
stream_create foo
stream_create bar
register_agent foo {} {
    stream_open bar
    on_stream_append {
        stream_append bar $argv ;#b
    }
}
register_agent bar {} {
        on_stream_append {
                puts $argv
                stream_relocate 1 ;#c
        }
        on_stream_relocation {
            set my_loc [stream_location]
            puts "I am at $my_loc" ;#d
        }
}
stream_move foo 1
stream_move bar 2
```

Figure 2: A simple auto-reconfiguring reactive system scripted from a single point of control.

A distributed application is constructed by first specifying the communication end-points as MStreams and then attaching Agents to those end-points, that in turn attach Handlers for specific Events. A given MStream may have multiple Agents and each Agent may register Handlers for different Events, but each Agent may have only one Handler for a given Event. When an Event occurs, the appropriate Handlers in each Agent are concurrently and independently invoked with appropriate arguments. Handlers are typically registered on Agent initialization and may be dynamically changed during execution.

An application built using our Middleware, may be thought of as consisting of two distinct parts - an active part and a reactive part. The reactive part consists of MStreams and Handlers. The active part or *Shell* lives outside the Middleware and drives it. A Shell may connect to the Middleware and issue requests and may exit at any time. The reactive part is persistent.

Figure 2 shows an example script that instantiates a simple distributed system that resides at *Sites* 1 and 2. A message is sent to the MStream called *foo* by the `stream_append` command issued via the external Shell (#a in Figure 2) . The MStream called *foo* receives the message "Hello world" and sends it to the MStream called *bar* (#b in Figure 2) which outputs the message via its handler and then moves MStream *bar* to Site 1 (#c in Figure 2). The arrival handlers run when the MStream *bar* arrives at *Site* 1, printing the string "I am at 1" to the console at Site 1 (#d in Figure 2).

In Figure 2 the script labelled "External Input" in is the Shell and MStreams and their registered handlers are the reactive parts.

2.1 Dynamic Extension and Re-configuration

An application built on our Middleware may be dynamically extended and re-configured in several ways while it is in execution (i.e., while there are pending un-delivered messages). First, an Agent can dynamically change the handlers it has registered for a given Event. Second, new Agents may be added and existing Agents removed for an existing MStream. Third, new MStreams may be added and removed. Fourth, new Sites may be added and removed, and finally, MStreams may be moved dynamically from Site to Site.

When an MStream moves from one Site to another, it (logically) moves the code of all of the Agents attached to it to the new Site along with whatever state they have placed in their *briefcase* structures. We say an Agent "visits" a Site when its MStream visits the Site. When an Agent first visits a Site, its initialization code executes there and when an Agent is killed, its (optional) *Finalization Handler* runs at each location that has been visited by it. Agent state (consisting of global state variables and code) is replicated at each site that it visits until the Agent is destroyed. On Agent destruction, the Handlers that it has registered are de-registered, and the interpreter and state variables are freed at each Site that it has visited. We assume that Sites may fail or disconnect during execution. Site failure does not imply destruction of the MStreams that reside there. Failure processing is described in Section 2.3.

The Agent's briefcase specifies a consistency requirement for moves. When an Agent moves from Site to Site only the elements in the briefcase are copied from the source execution environment to the target. The remainder of the global state remains unaffected (and cached) at the source site of the move. On successful completion of a move, the *Arrival Handlers* of the MStream are invoked at the new Site where the MStream has moved.

Handlers may move the MStream to which they are attached and also may move other MStreams around as well as create and destroy MStreams. Handlers may also exit - destroying the Agent in which they are housed and may also destroy other Agents. Such actions may also be initiated from an external Shell. Re-configuration may be contained by using appropriate policy handlers.

All changes in the configuration of an MStream such as MStream movement, new Agent addition and deletion, and MStream destruction are deferred until the time when no Handlers of the MStream are executing. We call this the *Atomic Handler Execution Model*. Message delivery order is preserved despite dynamic reconfiguration, allowing both the sender and receiver to be in motion while asynchronous messages are pending delivery.

Applications built using Mobile Streams can be extended from multiple points of control; any handler or Shell that has acquired an open MStream handle, can attempt to re-configure or extend the reactive part of the system and these actions can occur concurrently. While this adds great flexibility, it also raises several security and stability issues. We provide a means of restricting system reconfiguration and extension using control Events that can invoke policy Handlers. These policy Handlers may be registered only by privileged Agents as described below. We follow a discretionary control philosophy by providing just the mechanism and leaving the policy up to individual applications. Controls may be placed via policy Handlers at a session-wide level, site-wide level and at the level of individual MStreams for various security-relevant Events.

In summary, our security mechanisms are based on the following three principles:

Session-wide control: We have built mechanisms to place session-wide controls over extension and reconfiguration via a centralized *Session Leader* MStream.

Site-specific control: Each site may specify security policies via a *Site Controller* MStream. Site-specific policies may be used to grant or deny MStream entry to a site and to sand-box the incoming MStream handler's code by using safe-Tcl mechanisms.

Stream-specific control: The MStream itself is regarded as an extensible entity to which Agents can be attached and detached. It can carry its own policy Handlers to allow or disallow such actions, as determined by its *Stream Controller* Agent.

Our security implementation works as follows: Messages are classified as data messages and control messages. Data messages are delivered directly to the MStream. Control messages are messages that can change the configuration of the distributed system. These are routed first through the trusted intermediary Session Leader that can accept or deny these actions via its policy handlers, and then through the Site Controller which may again accept or deny the action and finally through the Stream Controller for MStream-specific actions. We invite the interested reader to look at [9] for more details.

The mechanisms described above permit us to build highly flexible and extensible distributed reactive systems that are able to extend and re-configure themselves, and also to place constraints on how the system can be re-configured and extended.

2.2 Message Delivery

Within our Middleware framework, point-to-point messages are delivered using an in-order sender-reliable delivery scheme built on top of UDP. All messages are consumed in the order they are issued by the sender despite failures and reconfigurations. These ordering and delivery guarantees make it simpler to design distributed systems.

In our scheme, the sender of the message is responsible for re-transmitting the message on timeout. We use a sliding-window acknowledgement mechanism similar to those employed by TCP. The sending Site buffers the message and computes a smoothed estimate of the expected round-trip time for the acknowledgment to arrive from the receiver. If the acknowledgment does not arrive in the expected time, the sender re-transmits the message. The sender keeps a window of unacknowledged messages and controls flow by dynamically adjusting the width of this window depending upon whether an ACK was received in the expected time or not. Thus far, our description is similar to the mechanisms employed by TCP. We have implemented our own protocol, rather than just use TCP, because TCP does not address certain conditions such as failures above the transport level and dynamic movement of the communicating end-points.

As previously described, an application can be dynamically reconfigured at any time with both the sender and receiver moving. When movement of an MStream occurs, a *Location Manager* is informed of the new Site location where the MStream will reside. This information needs to be propagated to each Handler or Shell that has opened the MStream.

When the target of an Append moves, messages that have not been consumed have to be delivered to the MStream at the new Site. There are two design options in dealing with this problem - either forward un-consumed messages from the old Site to the new Site or re-deliver from the sender to the new Site. Forwarding messages has some negative implications for reliability. If the Site from which the MStream is migrating dies before buffered messages have been forwarded to the new Site, these messages will be lost. Hence, we opted for a sender-initiated retransmission scheme. The sender buffers the message until it receives notification that the handler has run and the message has been consumed, re-transmitting the message on time-out.

When an MStream moves it takes various state information along with it. Clearly, there is an implicit movement of handler code and Agent execution state (via the briefcase), but in addition, the MStream takes a state vector of sequence numbers. There is a slot in this vector for each "alive" MStream that the MStream in motion has sent messages to or received messages from. Each slot contains a sent-received pair of integers indicating the next sequence number to be sent or received from a given MStream. This allows the messaging code to determine how to stamp the next outgoing message or what sequence number should be consumed next from a given sending MStream.

2.3 Handling Failures

A failure occurs when the Site where the MStream resides fails or disconnects from the Session Leader. Each MStream is assigned a reliable *Failure Manager* Site. When a such a failure occurs each of the MStreams located at the Site that has failed are implicitly relocated to its Failure Manager Site where its Failure Handlers are invoked. Failures may occur and be handled at any time - including during system configuration and reconfiguration. Pending messages are delivered in order, despite failures. A message is considered "consumed" only after all of the Append handlers execute at the target MStream for that message. (If none exist the message is discarded at the recipient). If the Site housing an MStream should fail or disconnect while a message is being consumed or while there are messages that have been buffered and not yet delivered, re-delivery is attempted at the MStream Failure Manager. To ensure in-order delivery in the presence of failures, the message is discarded at the sender only after the Append Handlers at the receiver have completed execution and the ACK for the message has been received by the sender. This is different from TCP where the receiver ACKs the message immediately after reception (and not after consumption as we require). After a failure has occurred at the site where an MStream resides, a failure recovery protocol is executed that re-synchronizes sequence numbers between communicating MStreams that involve the failed MStream. Each of the potential senders is queried to obtain the next expected sequence number. FIFO ordering can be thus be preserved despite the failure.

3 Implementation

We have implemented the Mobile Streams model in a toolkit we call AGNI [3] . AGNI is a multi-threaded TCL extension that uses the thread-safety features of TCL 8.1 and consists of roughly 23,000 lines of C++ code. In this section, we give highlights of the implementation some initial performance results.

Each workstation that wishes to participate in the distributed system runs a copy of an *Agent Daemon.* A distinguished Agent Daemon houses the Session Leader and is in charge of accepting or rejecting new Agent Daemons. This Daemon also serves as a Location Manager and Failure Manager for all MStreams in the Session. Each Agent Daemon has a unique identifier that it obtains from the Session Leader. Each Agent Daemon maintains a connection with the Session Leader Agent Daemon. Conceptually, the arrangement is as shown in the figure 3.

Each Agent has a TCL interpreter and thread of execution that is used by the Handlers that it registers. These resources are created for an Agent at a Site on its the first visit to the Site and remains allocated until the Agent (or the MStream to which it is attached) is destroyed. When a new Agent is added to an MStream, its code is propagated and initialized on the first move of the Agent to a previously unvisited Site, and remains cached there until it is destroyed. Provided an MStream has visited a Site previously, and no new Agents have been attached since its last visit, MStream movement simply consists of moving the state information in the briefcase (see Section 2) of each Agent of the MStream to the new Site and concurrently invoking each *on_stream_arrival* Handler.

Except for the case when the MStream is co-located with the Site from where the message originates, all control Events destined for an MStream (e.g. creation, relocation, new agent attachment) are delivered through the Session Leader Agent Daemon via the TCP connection that

[3]"AGents at NIst" (also Sanskrit for fire)

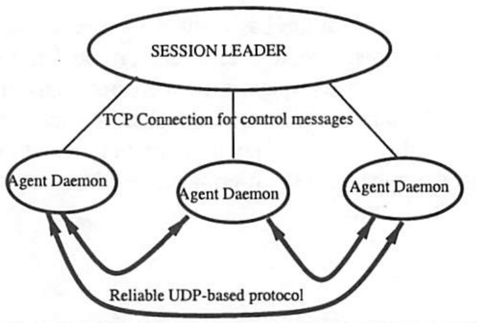

SESSION LEADER
TCP Connection for control messages
Agent Daemon
Agent Daemon
Agent Daemon
Reliable UDP-based protocol

Figure 3: Each Site runs an Agent Daemon that is connected to the Session Leader. The Agent Daemon is Multi-threaded with one thread per agent. The Session Leader maintains location and cache information.

each Agent Daemon maintains with it. The Session Leader Agent Daemon also acts as a Location Manager, keeping track of where each MStream is located and is hence able to re-direct messages to the location of the MStream. Sending all control Events through the Session Leader is a simple means of achieving a global ordering on control messages. The negative aspect of this design is that the Session Leader has the potential of becoming a bottleneck. However, we expect the number of control messages to be much smaller than the number of data messages (appends) processed by the MStream and hence do not consider this a serious limitation at present. In our future work, we plan to alleviate this problem by replication of the Session Leader. The Session Leader Daemon also manages the tracking information for the code and state cache described previously and is charge of propagating code to previously unvisited locations. As all Agent code is registered at the Session Leader and propagated from there, this simplifies the trust model to pair-wise relationships between each site and the Session Leader. provided all parties trust the Session Leader.

Appended data messages are delivered to the destination MStream directly without going through the Session Leader. Thus the Session Leader is not a bottleneck for data message delivery.

4 An In-Situ Simulation Environment

Estimating the detailed behavior and performance of a distributed system is hard. There are several degrees of variability and the interaction between physical effects is often difficult determine. Further, bugs - especially timing related ones can be quite difficult to reproduce in physical test-bed environments. In order to address these issues, we have developed an in-situ simulation environment that enables tuning, debugging and performance estimation of both the AGNI runtime system and applications.

Our approach system wraps a simulated environment around the actual AGNI system and application code using the CSIM [11] simulation library. We replace thread creations, locking and message sends and receives with simulated versions of these but leave the rest of the code unmodified. We have used the simulation for debugging and performance tuning the system as well as for testing the performance of applications built on top of our system. Figure 4 shows the simulation code for the introductory application presented in section 2. As can be seen from the example, the simulation and the actual system script look quite similar, with the exception of the parameters at the top and some new commands to create simulated processes and shells. The simulation runs as a single process whereas the actual system consists of multiple communicating processes. The simulation contains various "tweaking" parameters that have to be adjusted to match reality. These parameters include the message latency, packet drop percentage and simulated delays corresponding to code execution time. The goal in tuning the simulation is to adjust these parameters to make the simulation match the behavior of the real system for the quantities of interest. Presumably, we can match these over some simple scenarios and then try more complex ones, having some assurance of the validity of results. One can get a good idea about what delays are significant by looking at the gprof execution trace for the actual system.

A simulation is, however, only good to the extent it matches reality. Fitting the simulation to reality involves several cycles of adjusting performance parameters and re-testing the simulation. There is a large degree of variability in the performance of the actual system. We aim to make the output of the model fall within one standard deviation of the actual system for the quantities of interest. To test if this is feasible, we tested some simple scenarios. In both the real and simulated environments, we emulated packet drop by randomly dropping packets at the receiver. The quantities of interest that we would like to match are the throughput of packets and the number of packets sent by the sender for each packet consumed by the receiver (packet ratio). While we are still in the process of tuning the simulation, our initial results are encouraging. Figure 5 shows the message count performance of the real system and simulated system for two fixed end-points.

```
set drop  10
set m0 [ machine m0 ] ;#1
set m1 [ machine m1 ]
set e0 [ create_engine $m0 -d $drop ] ;#2
set e1 [ create_engine $m1 -d $drop ]
sim_set_send_latency m0  .0005 ;#3
sim_set_send_latency m1  .0005
sim_set_execution_time "Tcl_FindHashEntry" 0.000015 ;#4
sim_set_execution_time "Tcl_NextHashEntry" 0.000015
sim_set_execution_time "Arrival" 0.000015       ;#5
create_shell $e0 {
  stream_create foo
  stream_create bar
  register_agent foo {} {
        stream_open bar
        on_stream_append {
             stream_append bar $argv
        }

  }
  register_agent bar {} {
        on_stream_append {
                puts "$argv"
                stream_relocate 1
        }
        on_stream_relocation {
             puts "I am at [stream_location] at [sim_clock]"
             conclude_sim
        }
  }
  stream_move foo 0
  stream_move bar 1
  stream_append foo "bar"
}
sim 100000
```

Figure 4: Simulation script for introductory example.

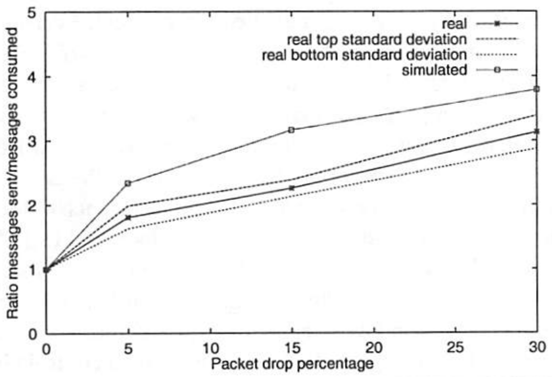


Figure 5: Simulated versus actual packet ratio for fixed end-points.

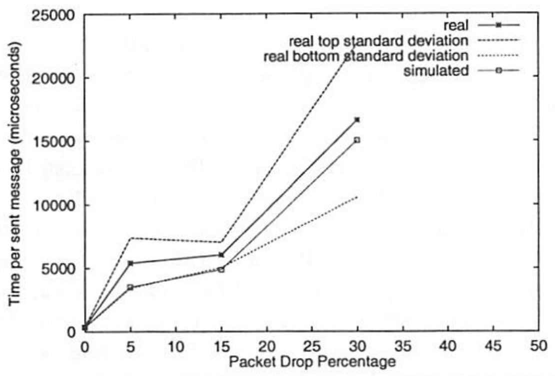


Figure 6: Simulated versus actual message consumption time for fixed end-points.

5 Application Sketches

In building applications using our infrastructure we adopted a problem-driven approach in mapping AGNI capabilities to prototype solutions. For example, we started with the assumption that we would use mobility only to the extent that it simplified the application design or enhanced performance in some fashion, rather than adopt the approach that mobility is a feature whose utility needed to be demonstrated. The remainder of this section outlines the design of two applications. The interested reader is referred to [9] for additional examples.

5.1 Distributed Data Combination

Consider a distributed experiment where data is being gathered at multiple sites and a query involves picking up data items from each location and combining the data to produce a composite result. Such an application may be structured as a master server front end and a multiple slave back ends. The front end gets the query and farms it out to to each of the participating sites. The slave sites process sub-queries locally, gathering results and returning them to the master site which then returns the combined result to the remote caller. If the combination operation is an involved one such as a database join, this could place an excessive burden on the master. Alternatively, this operation could be offloaded to the client or one of the slaves. When the query is received by the master, it creates an MStream at the client or one of the slaves to receive data from the data sources and and process the join.

If the data can be shipped incrementally from the data source, and results can be produced incrementally, the operation that receives results can be moved dynamically between the slaves and the client depending on available bandwidth and other machine resources. Such techniques are useful for optimizing dynamic query execution in client-server database systems. The ordering guarantee provided by MStreams ensures that the incremental join results are received in order by the join operator and the output operator regardless of the physical location of the join operator. In particular the join operator could be dynamically moving between sites as the join is being processed. Several positioning strategies may be considered in dynamically moving the join operator around. Some of these strategies are considered in [7] where we consider the more complex case of a join tree and adapt the operator placement to bandwidth variations. Figure 7 shows the overall organization of the system.

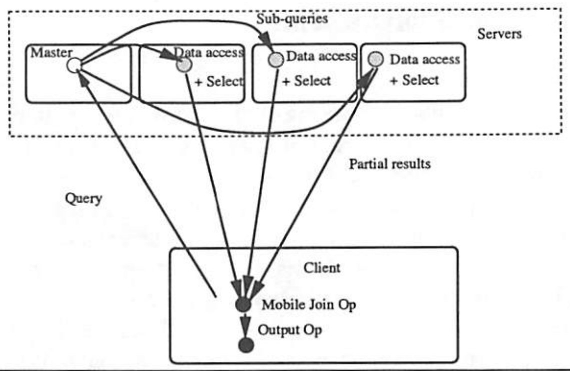


Figure 7: Adaptive database query execution. The join operator may be dynamically repositioned while the join is in progress.

5.2 Collaborative Annotation of Experimental Data

Although we have described our tool primarily as a means of scripting distributed applications, there is no requirement that the MStreams reside on physically separated machines. In section we describe SMAT - A Synchronous Multimedia Annotation Tool. SMAT was designed to be part of a scientific collaboratory for use in a robotic arc welding research project at NIST [13].

The scenario is as follows: Data is produced by sensors in various parts of a welding system and welding cell controller. Data from these sensors can have different media types - for example, video, audio and discretely sampled current and voltage. The primary functional requirement for SMAT was to develop a tool that supports the capability synchronize and play back the captured multi-media data after the weld is complete and provides a means to stop the playback at any point in time and enter annotations. After the annotation session is complete, the entered annotations are uploaded to an server for other users to view and annotate. During subsequent sessions, the media and the annotations are played back in synchronous fashion. Annotations appear in the annotation window corresponding to the relative time at which they were entered.

A secondary requirement for SMAT was to support real-time collaboration in the tool. Using this capability, users may effectively have partial control over each other's tools in order to share the same view of the multimedia data.

To meet these requirements, SMAT was designed as a control and integration framework that exploits existing tools to play specific media types. We started with the assumption that each tool to be controlled exports an API or mechanism (such as COM) that permits it to be controlled from another process. The tools are all tied together using a common *control bus* implemented using MStreams. The idea of the bus is much the same as the idea of a bus in computer hardware. Components are tied together by plugging them into the software bus in the same fashion as cards are plugged into a hardware bus. The components in this case are slave processes that play the different multimedia files. In order to use such an approach, the interfaces to the tools under control must be made uniform. To achieve this uniformity we wrap a controller script around each tool. For example, we can use XANIM as a tool that plays video under UNIX. XANIM takes external input via property change notifications on an XWindow Property. If we use a Microsoft tool, it may export COM interfaces for external control. In general, each tool may have its own idiosyncrasies for external communication. We encapsulate these via a software driver wrapper that hides the communication complexities from the control layer and registers standardized callbacks with the control layer. This is modeled after a device driver in an operating system that would register *read*, *write*, *ioctl*, *open* and *close* callbacks. The callbacks in our case include a *start* interface, a *stop* interface, a *quit* interface, a *timer tick* interface and a *seek* interface. These get called from the controller at appropriate times. It is up to the driver to communicate with the slave tool if need be on each of these calls. To enhance usability, we need the look and feel of a single tool rather than several individual tools. For this, we use Tk window embedding. Each tool that has a embedable top level window is embedded in a common canvas. The overall tool is controlled by the user via a control GUI that also sends events through the control bus. The architecture is shown in Figure 8.

There are several advantages to structuring a tool in this fashion:

Distributed Control Each tool is controlled by a separate AGNI Agent that implements its driver. The driver reacts to events that can be generated from anywhere in the distributed application. For example, the slider tool can append messages to the controller that re-distributes these events as *seek* events to each of the tool drivers. If the multimedia tools support random seeks, they can respond to such seek requests and position their media appropriately, thereby giving the ability to have both real-time and manually controlled synchronization. If we wanted to share the slider, in a synchronously collaborative fashion, this seek input simply needs to originate from another machine rather than the local slider. The control inputs could also come from another collaborative environment and in-

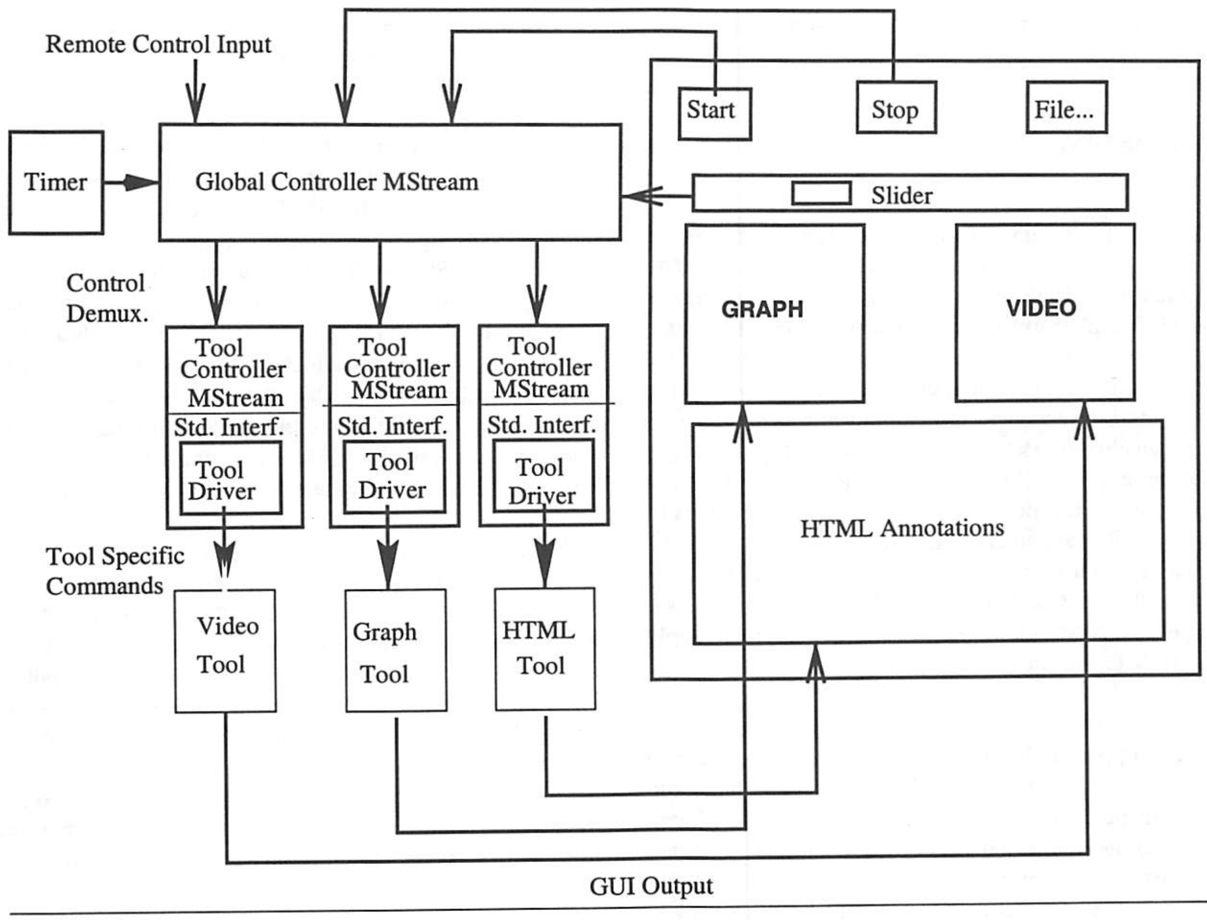


Figure 8: SMAT: A composite annotation tool with a distributed control bus. Each media type is handled by a separate tool with its own driver. An MStream-based event-bus is used to tie together the tools and provide a means to selectively export controls.

deed we have used this approach to integrate the tool in with the Teamwave client [12].

Isolation of Components Each tool runs in its own address space. Thus, a misbehaving tool cannot bring down the application. Failures are easy to isolate and fix. We can utilize off-the-shelf tools for media handling and annotation whenever such tools are available. For example, in our Windows NT version of the tool, we use the COM *IWebBrowser2* interfaces to Windows Explorer and drive it as an external tool to allow us to browse annotations. We use the COM *IDispatch* interface to Microsoft Word to bring up an editor to enter annotations.

Modularity and Extensibility: As all drivers export uniform interfaces, it is easy to add new media types. We simply build a driver to encapsulate the interface to the tool and plug it into the bus.

A practical issue that arises in this design is how to deal with cleanup. When the main interface exits or is killed the entire tool including all its components should be terminated. To deal with this problem, we use the *client_attach* and *client_detach* events for which the Site Controller MStream may register Handlers. These handlers are executed when a client attaches or detaches from the daemon at a given site. It can issue messages to the other tool controllers MStreams to exit the tools that they control.

It may be a concern that the decomposition of the system into processes degrades performance. Our experience was that degradation in performance is not unacceptable. The system appeared to behave well even on a slow machine (130 MHz) running windows NT.

We are also working on a data collection facility that will monitor the system, gather data and populate ftp repositories with the data after experiments are completed.

6 Related Work

Tcl DP [10] is the most popular extension for distributed scripting. Our first point of comparison is with this system. In contrast to Tcl DP that is RPC oriented, our system is intended as a platform to script distributed event-oriented applications. We rely on one-way messages to support this. In Tcl DP, messages are round-trip and the sender cannot proceed until the recipient has completed processing. Our system can also support synchronous (round-trip) messages where the sender blocks until the append handler at the target completes execution and hence we can do the kinds of things Tcl DP is aimed at doing. However, we expect most applications built using our system to be one-way message oriented. It is interesting to note that in our system, we can move the server in response to an RPC before the reply comes back to the client.

Our framework and toolkit is related to several other systems that support mobility. In contrast to other research in Mobile Agents, our approach has been to treat mobility and Mobile Agent technology as an enhancement to distributed scripting rather than as a means of supporting disconnected operations. Consequently, we have concentrated on typical distributed systems issues such as location tracking, message passing and failure handling. This distinguishes and separates our work from the other work in this area. Tcl provides an ideal platform for building mobile agent systems and there have been a few such systems that have gained popularity. Agent Tcl [3] supports a generalized mobility model where migration is allowed at arbitrary points in execution of the mobile code. This provides greater flexibility and perhaps a more natural programming model than we provide. However, this approach suffers from a few shortcomings. First, it requires modification of the core Tcl distribution - something that is difficult to keep up with over the long run. Unrestricted mobility makes support of fault tolerance and reconfiguration harder to achieve. In contrast, our system restricts mobility and other state changes to handler boundaries and treats handlers as atomic units of execution. By providing such a clean execution model, we simplify the system design and implementation while increasing slightly the burden of the developer using our system. Previously, we had developed a system called *Sumatra* that supports unrestricted mobility for Java applications by modification of the Java Virtual Machine [8] and many of the design decisions in this system are influenced by the experience gained in the Sumatra exercise. TACOMA [5] is another Tcl-based mobile agent system that adopts a programming model similar to ours. However, there are some basic difference as outlined below.

In this work, we proposed direct communication (reliable message passing) between Mobile Agents. In our system *on_stream_append* Handlers (analogous to "Agents" in other systems) pass one-way messages to each other reliably (via MStreams) rather than meeting to exchange messages, using a blackboard or other RPC-like mechanisms. Cabri et. al. [1] argue that this is not such a good idea for several reasons which make sense in the context of free-roaming disconnected agents. Our system is oriented towards building re-configurable distributed applications rather than supporting free-roaming autonomous entities and hence several of their concerns do not apply.

Aglets [6], Voyager [2], and Mole [14] are Java-based systems that follow a programming model similar to ours. However, our system differs from these systems in the following important ways: (1) Our design philosophy is to incorporate reconfiguration into a distributed system building toolkit rather than support disconnected operation as the fundamental design goal, (2) We have incorporated a peer-to-peer reliable, resilient message delivery protocol that none of these other systems offer and (3) We have a means of restricting system re-configuration and extension using policy Handlers that separate global (system-wide), and local concerns.

Dynamic re-configuration of distributed systems has been considered by Hofmeister and Purtilo [4] using a software bus approach. Their system supports dynamic changes to modules, geometry and structure of a distributed system. However, failure processing and asynchronous message delivery during reconfiguration is not considered.

7 Conclusions and Future Work

In this paper we have presented the motivation and design of a Middleware framework that uses mobility to simplify distributed scripting. We presented examples to illustrate the use of our system. Our system may be downloaded from from http://www.antd.nist.gov/itg/agni/.

Our plans for extending the Middleware is concentrated in three areas. We will incorporate reliable multicast primitives in our system whereby an MStream can communicate with a group of MStreams. As in the unicast case, both the sender and the recipients can be in motion while messages are being delivered. Second, we intend to make our location tracking scheme more robust and scalable by using replication and multicast. Third, we will build persistence at the location manager so that the system can be stopped and restarted without loosing all the MStreams and data. Finally, we intend to continue building applications - especially in the domain of mobile computing and distributed testing.

8 Acknowledgments

The authors acknowledge and appreciate the contributions of Virginie Schaal (NIST), Virginie Galtier (NIST), Laurent Andrey (LORIA, Fr.) and Anurag Acharya (UCSB) to this project. We thank Kevin Mills, Mark Carson and Craig Hunt of NIST, for reading this paper and making useful suggestions to improve its content, readability and presentation.

References

[1] G. Cabri, L. Leornardi, and F. Zambonelli. Coordination in mobile agent systems. Technical Report DSI-97-24, Universita' di Modena, October 1997.

[2] Object Space Corp. Voyager white paper. http://www.objectspace.com/voyager.

[3] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop - Monterey CA*, July 1996. http://www.cs.dartmouth.edu/ agent/papers.html.

[4] Christine R. Hofmeister and James M. Purtilo. Dynamic reconfiguration of distributed programs. In *11th. International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[5] Dag Johansen, Robbert van Renesse, and Fred B.Schnieder. An introduction to the TACOMA distributed system. Technical Report 95-23, University of Tromso, Norway, June 1995. *http://www.cs.uit.no/DOS/Tacoma/tacoma.webpages*.

[6] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998. ISBN 0-201-32582-9.

[7] M.Ranganathan, Anurag Acharya, and Joel Saltz. Adapting to bandwidth variations in wide-area data access. In *International Conference On Distributed Computing Systems*, pages 498–506, May 1998.

[8] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *USENIX Winter Technical Conference*, jan 1997.

[9] M. Ranganathan, V. Schaal, V. Galtier, and D. Montgomery. Mobile streams: A middleware for reconfigurable distributed scripting. In *Agent Systems And Architectures/Mobile Agents '99 (to appear)*, October 1999.

[10] Brian Smith, Tibor Janosi, and Mike Perham. Tcl dp. http://www.cs.cornell.edu/Info/Projects/zeno/Projects/Tcl-DP.html.

[11] Mesquite Software. Csim-18 simulation library. http://www.mesquite.com.

[12] Teamwave Software. Teamwave collaborative toolkit. http://www.teamwave.com.

[13] Michelle Steves, Wo Chang, and Amy Knutilla. Supporting Manufacturing Process Analysis and Trouble Shooting with ACTS. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* , June 1999. http://www.mel.nist.gov/msidstaff/steves.micky.html.

[14] Markus Straer, Joachim Baumann, and Fritz Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz.

Introducing QoS awareness in Tcl programming: QTcl

Roberto Canonico, Maurizio D'Arienzo, Simon Pietro Romano, and Giorgio Ventre
Dipartimento di Informatica e Sistemistica, Università di Napoli "Federico II", Napoli, Italy
{canonico, darienzo, sprom}@grid.unina.it ventre@unina.it

Abstract

A number of distributed applications require communication services with Quality of Service (QoS) guarantees. Among the actions undertaken by the Internet Engineering Task Force (IETF) with regard to the end-to-end QoS provisioning issue in the Internet, the definition of the Integrated Services (*IntServ*) framework plays a major role. According to this model, applications need to interact with network routers by means of a signalling protocol, RSVP. Even though special-purpose APIs have been defined to let applications negotiate QoS parameters across RSVP-capable networks, the integration of QoS negotiation mechanisms in the applications still remains an open issue. In modern programming, the Tcl scripting language plays a significant role, as it enables fast system prototyping by gluing basic components to build complex applications. In this paper we present QTcl, an extension of Tcl-DP which provides programmers with a new set of primitives, fully compliant with the SCRAPI programming interface for RSVP. We also present how QTcl has been used in an advanced VoD application to setup reservations in an IntServ network.

1. Introduction

Building global-scale distributed systems with predictable properties is one of the great challenges for computer systems engineering in the new century. Quality of Service (QoS) requirements will be critical for a large number of these systems, in particular for distributed applications whose performance depends mainly on the characteristics of the communication service provided by the networking infrastructure [1]. The global network par excellence is the Internet, with millions of users spread world-wide. Communication on the Internet is based on the connectionless IP protocol, which offers only a best-effort service. Hence, a great effort has been made in the past years to provide advanced communication services with QoS guarantees in IP-based networks.

The Internet Engineering Task Force (IETF) has defined an Integrated Services (*IntServ*) [2] framework to provide a service model that includes best-effort service, real-time service and controlled link sharing. According to this model, applications can, with the help of an appropriate signalling protocol like *RSVP (Resource reSerVation Protocol*) [3], request communication services with bounds on communication throughput or end-to-end latency. To do so, network routers need to implement special resource management policies and packet scheduling algorithms. However, to build distributed systems which benefit from the advantages of new networking services, we need to design QoS-aware applications, i.e. applications that know exactly their communication requirements and are able to interact with the network to negotiate the quality of the communication service. Hence, the need of defining ad-hoc APIs, which let applications issue per-stream resource reservations.

A large number of applications, in particular multimedia applications, consist of a set of pre-existing components (*building blocks*) glued together by a common GUI. To develop applications of this kind, scripting languages have proved to be better suited than system programming languages [4]. It is then reasonable to provide support for QoS into modern scripting languages. In this paper we present QTcl, an extension of Cornell's Tcl-DP. QTcl extends the Tcl-DP interpreter by providing a set of new commands, according to the SCRAPI application programming interface, defined by the IETF [5]. This API conforms to a simplified model, in order to reduce the complexity of the development of new QoS-aware applications.

The rest of the paper is organised as follows. In section 2 we briefly describe the *IntServ* model and the SCRAPI programming interface. In section 3 we present QTcl and the set of new commands. We show the use of QTcl commands in a simple application in section 4. In section 5 we illustrate how we have implemented QTcl, as an extension to Cornell's Tcl-DP. Finally, in section 6 we present a distributed VoD application, which uses QTcl to protect its data flows in a QoS-enabled internetwork.

2. QoS in the Internet

IP has been playing for several years the most important role in global internetworking. Its connectionless nature has proved to be one of the keys of its success.

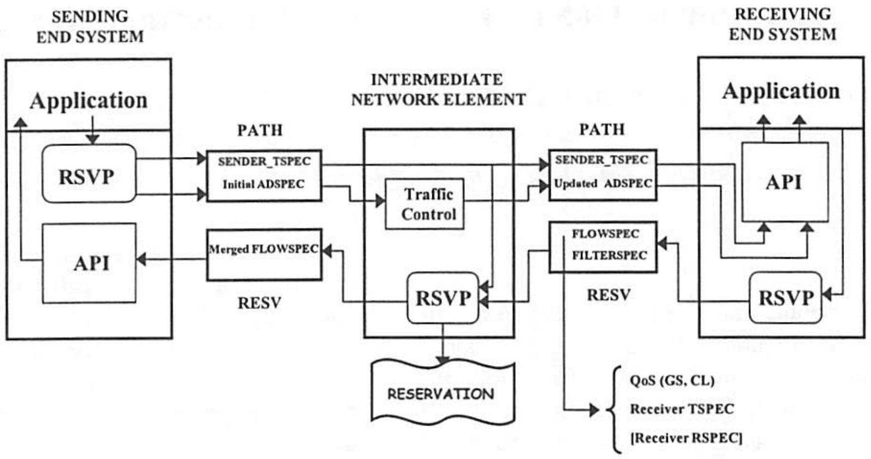


Figure 1: The use of RSVP objects during the resource reservation phase

Based on this assumption, the IETF Integrated Services working group has specified a control QoS framework [2] in order to provide new applications with the appropriate support. Such a framework proposes an extension to the Internet architecture and protocols which aims at making broadly available integrated services across the Internet.

The key assumption on which the reference model for integrated services is built is that network resources (first of all its bandwidth) must be explicitly managed in order to meet application requirements. The overall goal in a real-time service, in fact, is that of satisfying a given set of application-specific requirements, and it seems clear that guarantees are hardly achieved without reservations. Thus, resource reservation and admission control will be playing an extremely important role in the global framework. The new element that arises in this context, with respect to the old (non-real-time) Internet model, is the need to maintain flow-specific state in the routers, which must now be capable to take an active part in the reservation process.

Based on these considerations, the components included in the reference framework are a *packet scheduler*, an *admission control module*, a *packet classifier* and an appropriate *reservation setup protocol*. The first three modules together form the *traffic control interface* of the router. The reservation setup protocol is needed to create and manage state information along the whole path that a specific flow crosses between two network end-points. One of the features required to such a protocol is that of carrying the so-called *FLOWSPEC* object, that is a list of parameters specifying the desired QoS needed by an application. At each intermediate network element along a specified path, this object is passed to admission control to test for acceptability and, in the case that the request may be satisfied, used to appropriately parameterize the packet scheduler [6]. RSVP [3] is the resource reservation protocol recommended by *IntServ*.

Data treated by RSVP are of three natures, according to the entity that supplies them (sender and receiver) or modifies them (intermediate network elements). The information supplied by each sender, and conveyed in the *SENDER_TSPEC* object, concerns the type of traffic that it is going to generate. Receivers provide *FLOWSPEC* objects, which are built of two parts, *RECEIVER_TSPEC* and *RECEIVER_RSPEC* (both contained into a message called **RESV**). The former contains the traffic description the resource reservation should apply to, while the latter carries the service class to be used and the corresponding quality of service parameters.

Intermediate network elements, in turn, provide additional information such as available services, delay and bandwidth estimates, and additional service specific parameters. This information is contained in *ADSPEC* objects, and is used by the receivers to choose a service and determine the reservation parameters. *ADSPEC* and *SENDER_TSPEC* objects are both contained into a **PATH** message. Figure 1 shows the use of the defined messages during the resource reservation phase.

IntServ service classes define a framework for specifying services provided by network elements and available to applications, in an internetwork capable of offering multiple, dynamically selectable qualities of service. So far, two different service classes have been defined: *Guaranteed Service* (GS) [7] and *Controlled Load* (CL) [8].

In both cases, it is required that the sender provides, in *TSPEC* objects, a description of the traffic it is going to generate. Since traffic patterns are complex to describe, a worst case characterisation is provided (*traffic envelope*), according to a token bucket model [9]. Relevant parameters are the following:

- token bucket depth (b [Bytes]),
- average rate (r [Bytes/s]),
- peak rate (p [Bytes/s]),
- minimum policed unit (m [Bytes]),
- maximum datagram size (M [Bytes]).

A source conforming to such a description will generate, during any time interval of length τ, a number of bytes upper bounded by (Fig. 2):

$$A(\tau) = \min(b + r\tau, M + p\tau), \qquad \tau \geq 0$$

Guaranteed Service (GS) provides the clients data flow with firm bounds on the end-to-end delay experienced by a packet while traversing the network. It guarantees both bandwidth and delay. The GS emulates the service that would be offered by a dedicated communication channel between the sender and the receiver. Two parameters apply to this service: *TSPEC* and *RSPEC*. The *TSPEC* describes the traffic characteristics for which service is being requested. The *RSPEC* specifies the QoS a given flow demands from a network element. It takes the form of a *clearing rate* R and a *slack term* S. The clearing rate is computed to give a guaranteed end-to-end delay and the slack term denotes the difference between desired and guaranteed end-to-end delay after the receiver has chosen a value for R.

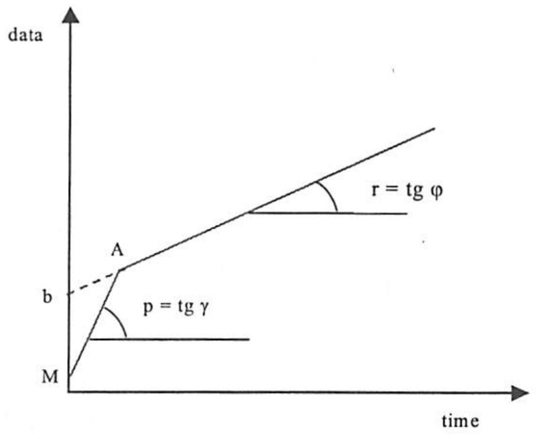


Figure 2: Traffic envelope for a source conforming to a (b, r, p) token bucket

Controlled Load (CL) service, on the other hand, may be thought of as a "controlled best-effort" service, i.e. a service with the same characteristics of a best-effort delivery over a not overloaded network. To avoid QoS degradation when the network load increases, CL relies upon admission control algorithms. CL is best suited to applications that have been developed taking into account the limitations of today's Internet, but are highly susceptible to overloaded conditions. A typical example is given by adaptive real-time applications, which have proved to work well when there is little or no load on the network.

2.1. QoS programming interfaces

In the framework we just depicted, the IETF has defined RAPI [10], an API compliant with the RSVP Functional Specification [3]. It is a user-level library written in C, which can be used by applications aiming at exploiting the QoS functionalities made available by a network reservation protocol like RSVP. RAPI calls let an application interact with a local RSVP daemon process, in order to establish a communication with QoS guarantees.

The RAPI interface is a first step towards the integration of communication services with QoS guarantees into applications; yet, its use is somewhat complex, since the application programmer must be aware of a number of parameters concerning the reservation. To cope with such problems, the IETF has proposed a simpler programming interface, layered on top of the RAPI and called SCRAPI [5]. SCRAPI provides only three functions:

- *Scrapi_sender*, to be used by the sender of a data stream associated to an RSVP session,
- *Scrapi_receiver*, to be used by the receiver, and
- *Scrapi_close*, to close an RSVP session.

Figure 3 shows the SCRAPI state diagram. A generic host starts in the **Closed** state. Then, it can act either as a sender or as a receiver or as a sender/receiver.

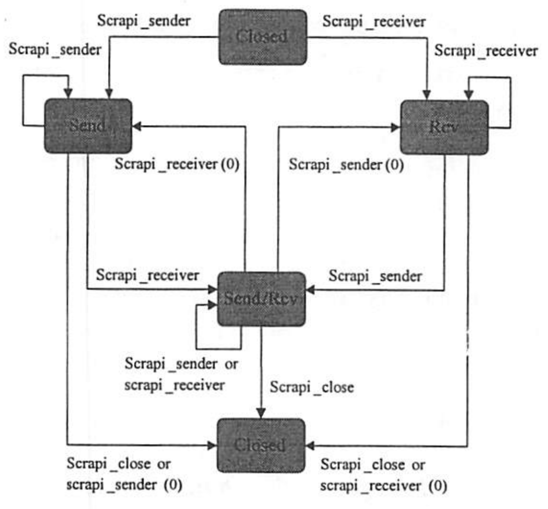


Figure 3: State diagram for the SCRAPI interface

SCRAPI differs from RAPI especially in the error-handling model. While RAPI requires the application programmer to implement a set of upcall routines, to handle asynchronous events and errors, this is not required anymore when using SCRAPI. Upcalls are replaced by a simplified "three colours" error model, which makes use of three different values (red, yellow, green), whose combinations let the application know the state of a reservation at a given instant in time. A reservation is said to be in the RED status if the transmission of **PATH** messages from the sender has not started yet, or **PATH** messages have not arrived at the respective receiver yet, or the system is currently in an error state. A YELLOW state indicates that a valid **PATH** message flow is present, but reservations have not been made by receivers. The transition to the GREEN state happens when the reservation is accepted. This strategy leads to a "light-weight" model, even if it imposes a number of constraints on the QoS negotiation process.

The service model used by the simplified interface builds the required RSVP objects in a way that is transparent to the end user. In particular, the object *SENDER_TSPEC* T:

[*token_bucket_rate* *token_bucket_depth*
min_policed_unit *max_datagram_size*]

is created based on the following assumptions:

- the average bandwidth, specified by the sender, is actually used as the token bucket rate (r) for the flow;
- the peak rate (p) is considered infinite;
- the token bucket depth (b) is assumed twice as the average bandwidth (and so two times r);
- the minimum policed unit (m) is 64 bytes;
- the maximum policed (M) unit is the greatest MTU associated to the IP interfaces available on the host.

As far as the receiver is concerned, the RSPEC object for the Guaranteed Service is built by simply setting the value of the clearing rate (R) to the average bandwidth and using a slack term (S) of zero. In the case of Controlled Load service, the receiver reserves as much bandwidth as the sender declares in the *SENDER_TSPEC* (r).

3. QTcl API

The SCRAPI programming interface has already been implemented as a C library, and used in modified Mbone tools. A support for QoS communication in Tcl applications, instead, was not available. Since we wanted to implement in Tcl a QoS-aware application for the distribution of multimedia documents, we have developed QTcl, an extension of the Tcl scripting language which implements the SCRAPI interface. QTcl provides the Tcl programmer with a set of new commands to create reservations in an RSVP-enabled internetwork. The new commands are shown below:

```
• dp_scrapiSender    dest_hostname
                     dest_port
                     source_hostname
                     source_port
                     bandwidth
                     protocol
• dp_scrapiReceiver  dest_hostname
                     dest_port
                     source_hostname
                     source_port
                     service
                     protocol
• dp_scrapiStatus    dest_hostname
                     dest_port
                     protocol
• dp_scrapiClose     dest_hostname
                     dest_port
                     source_hostname
                     source_port
```

Using these commands, it is possible to manage the whole process of reservation setup.

The bandwidth parameter must be expressed in Bytes/sec. The `service` parameter can be one of the following two values: `cl` indicating Controlled Load or `gs` indicating Guaranteed Service. Finally, the `protocol` parameter can be either `tcp` or `udp`.

`dp_scrapiSender` opens an RSVP session and starts **PATH** message transmission from source host to destination host. **PATH** messages are refreshed every 30 seconds.

`dp_scrapiReceiver` is invoked by a receiver in order to make a reservation request. The receiver specifies the desired QoS and class of service (Guaranteed Service or Controlled Load) according to the information contained into the **PATH** message. This request is forwarded to the sender across the network via a **RESV** message. After sending a **RESV**, the receiver waits for a confirmation of successful reservation from the sender for at most 10 seconds, as set by a specific timer; however, even in case of timer expiration the reservation process will go on.

`dp_scrapiStatus` allows to verify the current status of a session, according to the simplified error model available in the SCRAPI interface, i.e. it returns a RED, YELLOW or GREEN value according to the status of the RSVP session.

`dp_scrapiClose` is the function called to tear down an RSVP session, both in reception and in transmission.

4. A simple QTcl application

Figure 4 shows a simple application made of a sender process and a receiver process. The two processes should be executed on different hosts connected by an RSVP-enabled internetwork. The sender process invokes the `dp_scrapiSender` command, to start the transmission of **PATH** messages and then waits in a loop until the reservation is completed. The receiver process, instead, issues the `dp_scrapiReceiver` command to start the transmission of **RESV** messages and waits for the reservation to be completed. As soon as the reservation is completed, the sender starts transmitting UDP messages, 1480 bytes in length. The receiver, in turn, measures the time needed to receive a number N of such messages and estimates the received throughput. This simple application can be tested in order to verify that the achieved throughput is independent from the network conditions, as long as the routers implement an *IntServ* Guaranteed Service.

5. QTcl implementation

QTcl has been conceived as a tool for supporting the development of distributed applications with simple QoS requirements. As we did not want to reinvent the wheel, we felt that some useful features were already available in the Tcl-DP extension, developed at Cornell University [11]. In particular, we found the `dp_RPC` mechanism particularly suitable to support the receiver-initiated reservation mechanism of RSVP. Hence, QTcl has been developed starting from the original Tcl-DP source distribution. We then extended the Tcl interpreter by creating a set of C functions that implement the SCRAPI primitives.

Notice that SCRAPI is only a programming interface to access the RSVP service, which must be implemented by a proper operating system module. In UNIX-like systems, this is usually a daemon process, which runs with root privileges in the end systems. Our current implementation of QTcl is available for the SUN Solaris, FreeBSD and Linux operating systems. For these systems, an RSVP implementation is provided by ISI [12].

Sender.tcl

```
# Sender
#!/home/qtcl/bin/tclsh8.0

package require dp

set sender [dp_connect udp -host
143.225.229.105\
    -port 3000 -myaddr localhost -myport 5000]

dp_scrapiSender 143.225.229.105 3000 \
   143.225.229.116 5000 100000 udp

while {$status != "green"} {
 after 1000
 set status [dp_scrapiStatus 143.225.229.105\
                              3000 udp]
}
puts $status

set pkt ""
for {set i 0} {$i < 1480} {incr i} {
 append pkt x
}

puts "Press ctrl-C to interrupt ...."

while {1} {
 set lun [dp_send $sender $pkt]
}

close $sender
```

Receiver.tcl

```
# Receiver
#!/home/qtcl/bin/tclsh8.0

proc bench { N } {
  global receiver
  set count 0
  while {$count < $N} {
    set rcv [dp_recv $receiver]
    incr count [string length $rcv]
  }
}

package require dp

set receiver [dp_connect udp -myport 3000]
fconfigure $receiver -blocking 1

dp_scrapiReceiver 143.225.229.105 3000 \
              143.225.229.116 5000 gs udp

while {$status != "green"} {
 after 1000
 set status [dp_scrapiStatus 143.225.229.105\
                              3000 udp]
}
puts $status

set N  10485760
set T  [lindex [time { bench $N }] 0]
set BW [format "%2.3f" [expr $N*8.0/$T]]

puts "Elapsed time: $T microseconds"
puts "Estimated bandwidth: $BW Megabit/sec"

close $receiver
```

Figure 4: A simple client-server QTcl application

As for the Microsoft Windows operating systems, the implementation of QTcl is not straightforward, due to the different semantic of the Microsoft RSVP-API implemented as part of their Winsock2 API. However, we are currently investigating the possibility of undertaking the port of QTcl for this platform.

Figure 5 shows the global picture of a UNIX host running a QTcl application and interacting with an RSVP-enabled router.

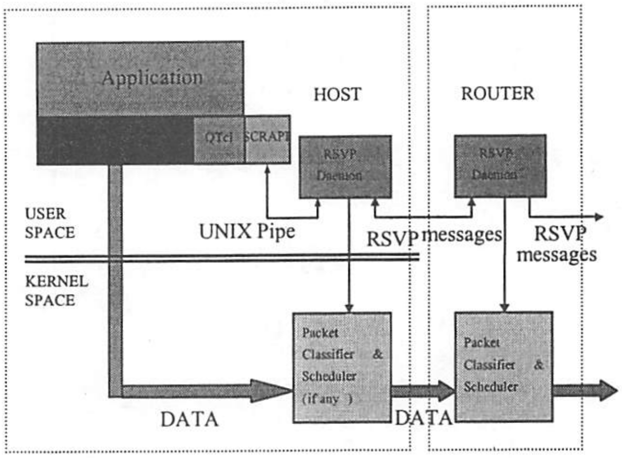


Figure 5: QTcl implementation in a UNIX host

5.1. Why Tcl-DP ?

Our latest version of QTcl has been developed from the release 4.0 of Tcl-DP. Tcl-DP communication services rely on different transport mechanisms: serial links, TCP, UDP, IP-multicast, and e-mail. Tcl-DP 4.0 is implemented as a loadable module, i.e. Tcl-DP commands are made available to the Tcl interpreter by means of the `package` command. Table 1 shows some of the Tcl-DP primitives that can be used to build a QoS-aware distributed application.

`dp_RDO`	Perform a remote procedure call without return value
`dp_RPC`	Perform a remote procedure call
`dp_MakeRPCServer`	Create a TCP RPC server channel
`dp_MakeRPCClient`	Create a TCP RPC client channel

Table 1: Tcl-DP commands

The RSVP protocol uses a receiver initiated approach. The practical consequences of this approach are different whether the application is based on a multicast or unicast communication. In a unicast based application (e.g. a Video on Demand system), a sender does not know in advance the address of the receiver. Hence, it can start sending **PATH** messages only after the receiver has declared explicitly its will of starting a session with resource reservations. To write such an application, the RPC mechanism provided by the Tcl-DP extension is extremely useful. Figure 6 provides an example of a unicast-based client application which uses a combination of the `dp_RPC` and `dp_scrapiSender` primitives to setup a reservation. The `dp_RPC` primitive invokes on a remote host a Tcl procedure, `ScrapiSndRsv`, which, among other things, in turn invokes the `dp_scrapiSender` primitive.

```
# Tell Video source to start sending PATH
msgs
if [catch {dp_RPC $sockV -timeout 60000 \
        ScrapiSndRsv \
        $obj(viAddrSrc,$urlSP) \
        $obj(viAddrDst,$urlSP) \
        $bwVideo $service} error] {
    catch {diva_CloseRPC $sockV}
    error "Server not responding: $error"
}
# Start sending RESV msgs upstream
set status ""
dp_scrapiReceiver \
        [lindex $obj(viAddrDst,$urlSP) 0] \
        [lindex $obj(viAddrDst,$urlSP) 1] \
        [lindex $obj(viAddrSrc,$urlSP) 0] \
        [lindex $obj(viAddrSrc,$urlSP) 1] \
        $service udp
while {$status!="green"} {
    after 1000
    set status [dp_scrapiStatus \
        [lindex $obj(viAddrDst,$urlSP) 0] \
        [lindex $obj(viAddrDst,$urlSP) 1]
udp]
}
```

Figure 6: Use of QTcl in combination with `dp_rpc` in a multimedia application to setup an RSVP session for a unicast stream.

6. A QoS-aware distributed multimedia application based on QTcl

To show the effectiveness of RSVP bandwidth management in a real application, we added the ability of making network resource reservations to DiVA, a distributed multimedia application developed by our research group. DiVA is capable of playing and controlling remote audio/video documents in streaming mode. Figure 7 shows the relevant data streams produced by the DiVA application among a streaming server host

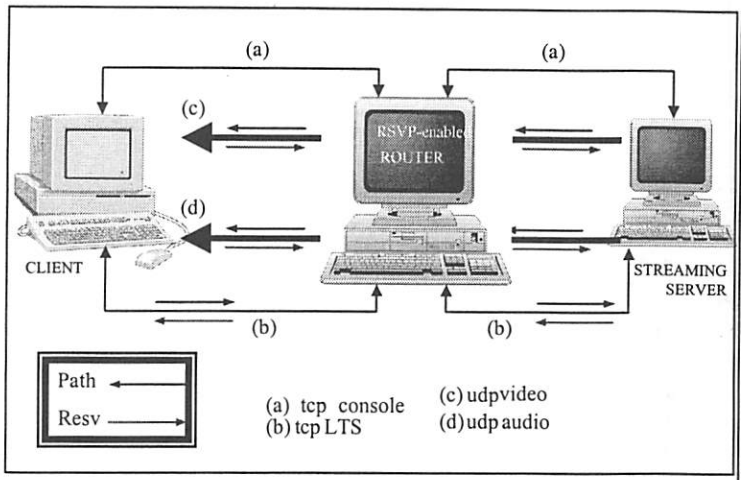


Figure 7: Data streams generated by the DiVA application and associated to RSVP sessions.

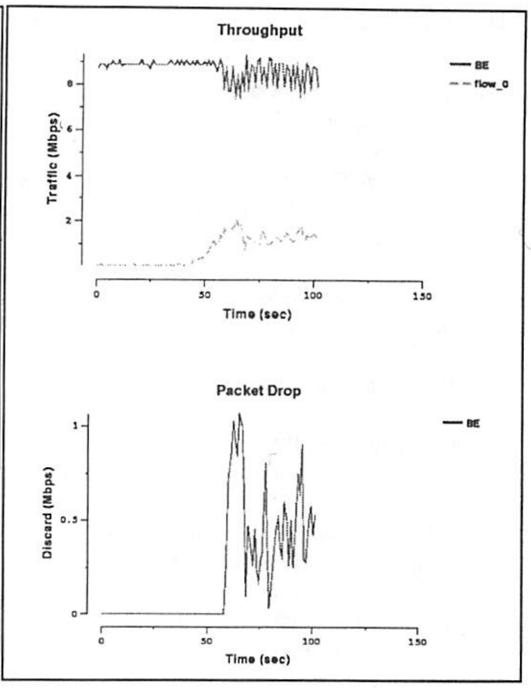


Figure 8: Streams behaviour with a reservation for the multimedia flows

and a client host. In particular, the UDP audio and video streams are transmitted downstream from the server on the right to the client on the left, while two TCP bi-directional streams are used to exchange control (console) and synchronization (LTS) information.

We tested the application in a testbed formed by two different Local Area Networks, connected by means of a WFQ router implemented in FreeBSD [6]. The router was connected to the first LAN through a 100 Mb/s Fast Ethernet card and to the second LAN through a 10 Mb/s Ethernet card. A host in the 10 Mb/s LAN acted as a client, while another host in the 100Mb/s LAN ran the DiVA video server. Hence, multimedia traffic flowed through the WFQ router.

To test the effectiveness of the traffic control mechanism implemented in the router, and the ability of the application to request the necessary Quality of Service, we generated a 9 Mb/s cross traffic stream among a pair of different hosts. Cross traffic and DiVA multimedia streams competed in the router for the 10 Mb/s bandwidth available in the Ethernet LAN.

In a first experiment, we did not make any reservation for the video/audio streams. In this case, multimedia traffic was not protected from the cross traffic and it had to share packet losses with it. Even though DiVA is capable of adapting the traffic generated to the available bandwidth, it was impossible to obtain a Quality of Service adequate for intelligible video and audio rendering in this case.

In a second experiment, we used RSVP to make a reservation for the flows generated by the video server, while cross traffic was still served as best effort.

In this case, Figure 8 shows that multimedia flows were fully protected from the best effort traffic, which started losing packets as soon as data streaming from the DiVA server began. This behaviour preserved a very good quality of video/audio rendering, in spite of the presence of cross traffic.

In our prototype, the bandwidth values used to setup reservations for the video and audio streams were determined empirically for each archived document, by observing the traffic produced by the application while streaming it. In a real-world application, these values should be retrieved by the client application in the form of *metadata* associated to the document.

7. Conclusions

An increasing number of distributed applications can benefit from the availability of improved communication services in RSVP-enabled IP internetworks, by acting in a proactive way, instead of passively adapting to the available QoS offered by current best-effort services. We believe that this support is helpful for a wide range of modern distributed applications. In this paper we have presented QTcl, a QoS control API which is compliant with the IETF SCRAPI interface, and has been designed as an extension of the Tcl scripting language. An implementation of QTcl for UNIX-derived operating systems is available on the web at: http://www.grid.unina.it/qtcl.

References

[1] K. Kavi, J.C. Browne, and A. Tripathi. "Computer Systems Research: The Pressure Is On". *Computer* , Jan. 1999, pp. 30-39.

[2] R. Braden, D.Clark, and S. Shenker. "Integrated Services in the Internet Architecture: an Overview". IETF *RFC 1633*, July 1994.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification". IETF *RFC 2205*, September 1997.

[4] J.K. Ousterhout. "Scripting: Higher-Level Programming for the 21st Century". *Computer*, March 1998, pp.23-30.

[5] B. Lindell. "SCRAPI - A Simple 'Bare Bones' API for RSVP". IETF Internet Draft draft-lindell-rsvp-scrapi-02.txt, Feb. 1999.

[6] R.D'Albenzio, S.P. Romano and G. Ventre. "An Engineering Approach to QoS Provisioning over the Internet". Lecture Notes in Computer Science no. 1629, Springer, May 1999, pp. 229-245.

[7] S. Shenker, C. Partridge, and R.Guérin. "Specification of Guaranteed Quality of Service". IETF *RFC2212*, September 1997.

[8] J. Wroklawsky. "Specification of the Controlled-Load Network Element Service". IETF *RFC 2211*, Sep. 1997.

[9] S. Keshav. "An Engineering Approach to Computer Networking". Addison-Wesley, 1997.

[10] R. Braden and D. Hoffman. "RAPI -- An RSVP Application Programming Interface - Version 5". IETF Internet Draft draft-ietf-rsvp-rapi-01.txt, Aug. 1998.

[11] M. Perham, B.C. Smith, T. Jánosi, and I.K. Lam. "Redesigning Tcl-DP". Procs. of the Fifth Annual Tcl/Tk Workshop, Boston, 1997.

[12] USC Information Sciences Institute (ISI), http://www.isi.edu/rsvp/release.html

CollabWiseTk:
A Toolkit for Rendering Stand-alone Applications Collaborative

Hemang Lavana Franc Brglez
CBL (Collaborative Benchmarking Lab), Dept. of Computer Science, Box 8206
NC State University, Raleigh, NC 27695, USA
http://www.cbl.ncsu.edu/

Abstract

Traditionally, a stand-alone client application is rendered collaborative for members of a team either by sharing its view or by re-writing it as a collaborative client. However, it may not be possible to anticipate in advance all preferences for collaboration, hence such a client may appear confusing to some of the team members.

We propose a novel client/server architecture for tk-based applications: rendering any stand-alone client collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The `CollabWiseTk` toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration. It re-defines the existing tk commands such that the entire tk widget set is rendered collaborative for use with multiple users.

We demonstrate the capabilities of the `CollabWiseTk` toolkit by readily rendering collaborative most of the Tk Widget Demonstrations, *distributed with the core Tcl/Tk. The toolkit is implemented in pure tcl and it ports to all platforms.*

Keywords: Internet, Collaboration, Groupware, Tcl/Tk, GUI.

1 Introduction

This paper is one of the two companion papers [1] that were initiated at the conclusion of the course on *Frontiers of Collaborative Computing on the Internet* (csc591-b, [2]).

A number of collaborative client/server architectures have been proposed to date. Principally, they deal with specific applications ranging from a shared calendar (The Electric Secretary) [3] to a shared white-board [4]; from collaborative visualization for health care [5] to collaborative editing of schematic diagrams [6]. An architecture that supports workflows of heterogeneous applications is described in [7, 8, 9]. Some architectures expect that the application has been written for a team of users, e.g. the GroupKit architecture [4]. Alternatively, multi-casting can render an application written for a single user collaborative, e.g. the REUBEN architecture [7, 8, 9]. There are disadvantages to both approaches: the need to write an application for multiple users, and the performance issues of multi-casting.

Most of the client applications today are as stand-alone applications. The traditional approach is to re-write it as a client for collaborative application. This can be a formidable task, especially when all possible preferences for modes of collaboration cannot be anticipated in advance. Such a client may turn out to be user-unfriendly or confusing for a particular team. Simple preferences, such as whether and when should the scrollbars track for all participating collaborators, or should separate scrollbars be provided (and color-coded) for each participant, are at the core of such issues [4, 10, 11].

In this paper, we propose a novel client/server architecture for tk-based applications: rendering any stand-alone client collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The `CollabWiseTk` toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration.

The paper is organized into following sections:

- Background and Motivation;
- `CollabWiseTk` Architecture;
- Inter-client Synchronization;
- `CollabWiseTk` Implementation;
- Testbed and Experiments;
- Software Evaluation;
- Software Availability and Status;
- Conclusions.

This research was supported by contracts from the Semiconductor Research Corporation (94–DJ–553), SEMATECH (94–DJ–800), and DARPA/ARO (P–3316–EL/DAAH04–94–G–2080 and DAAG55–97–1–0345).

2 Background and Motivation

Let us consider a very simple application which consists of a text widget with a vertical scrollbar. Such an application allows the user to type in text and the vertical scrollbar allows the user to browse the text information, when the size of the text widget is not large enough to display the entire text at the same time. This application can be very easily built using four lines of tcl code, as shown in Figure 1(a), and is a basic widget used by many complex applications that need functionalities such as syntax highlighted message display, text editing, and html display, to name a few.

Several possibilities exist even for a simple text widget that is rendered collaborative. Figures 1(b) - (e) shows various possible collaborative configurations of a text widget for two users Alice and Bob.

Figure 1(b) shows Alice and Bob sharing the same view of the text widget as well as the scrollbar. A centralized server ensures that both the views are synchronized at all the times. Possibilities of conflict arise when Alice and Bob both try to interact with the text widget at the same time. Such conflicts are typically resolved by some form of locking mechanisms, such as round-robin, first-come-first-serve, user-controlled token passing, etc, so that only one person can interact with the application at a time while the others are forced to watch.

Figure 1(c) shows a distributed implementation of a collaborative text widget that allows Alice and Bob to interact with their individual text widgets, while an event synchronizer dispatches these interactions to the other users. This mechanism also allows participating users to have different views of the same widget and occasionally 'glance' at the other widgets.

Figure 1(d) shows a configuration where both Alice and Bob get the same view of the text widget, but each has a personal edit cursor so that both can type in simultaneously without affecting the other. In the example shown, Bob has an edit cursor at the top, while Alice has an edit cursor at the bottom and hence both can work on different sections of the same text file. However, the associated scrollbar needs to be configured such that when Bob changes the scroll-view, Alice may not want to follow Bob's scrolled movement when she's editing, but may want to do so when merely watching Bob's interactions.

Figure 1(e) shows yet another configuration where the text widget is replicated once for every participating user. Therefore, Alice and Bob both have two text widgets - one where each can type in text and another where each can observe what the other is typing. Here again, Bob may prefer the scrollbars to be synchronized with Alice, while Alice may want to scroll independently Bob's text widget. This is equivalent to widely available chat tools or the Unix 'talk' utility.

All of the above examples can be easily implemented in tcl by re-writing the four lines of stand-alone tcl code. However, when this text widget is part of a more complex application containing several other widgets, each of which can themselves have their own numerous configuration possibilities, it becomes very difficult to anticipate in advance a suitable configuration preference. Such a collaborative application might turn out to be user-unfriendly or confusing for a particular team.

3 CollabWiseTk Architecture

The `CollabWiseTk` toolkit consists of two parts:

A synchronizing group server (SGS) that provides mechanisms for communication and synchronization among multiple-user client applications; and

A distributed collaboration client that provides mechanisms for inter-client synchronization among multiple-user client applications for effective collaboration.

The general architecture of the toolkit is shown in Figure 2. This architecture extends and complements the *Asynchronous Group Server Architecture* (AGS) in [1].

SGS is a tcl server that accepts socket connections from various collaboration clients. It has three types of repositories:

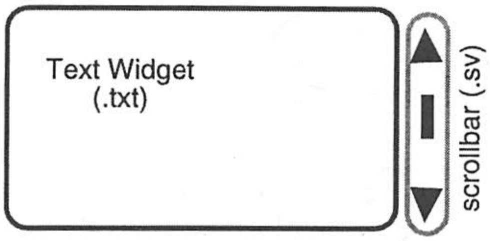
Tcl scripts and packages: various stand-alone tcl applications are deposited here and are available to users for collaboration;

Inter-client synchronization tables: different configuration preferences for a tcl application are stored in this tables; and

Registered users and access permissions: for security reasons, the latter maintains a list of registered users and their corresponding access privileges.

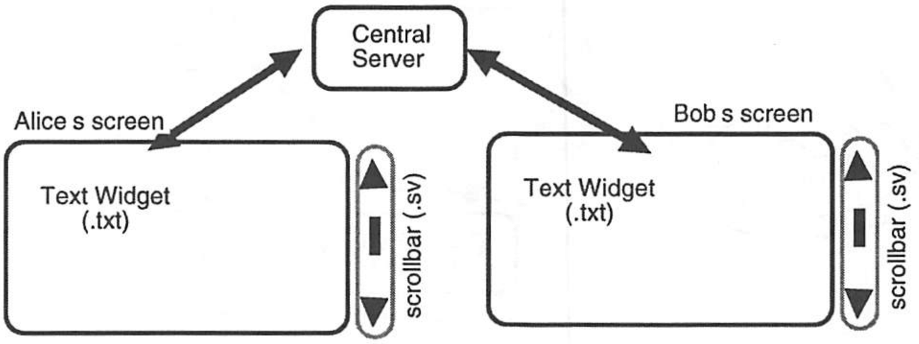
The collaboration client is installed on each user's machine and provides an interface to the user for collaborating with other users. Upon invocation, the collaboration client establishes a socket connection

(a) Simple text widget with a vertical scrollbar and its tcl code.

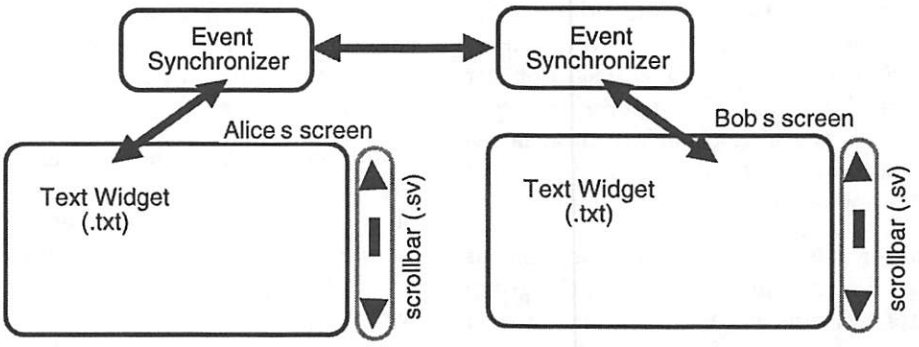


```
text .txt  -yscrollcommand ".sv set" \
    -height 15 -width 50
scrollbar $scrv -command ".txt yview" \
    -orient vertical
pack .txt -fill both -side left -expand 1
pack .sv -fill y -side left
```

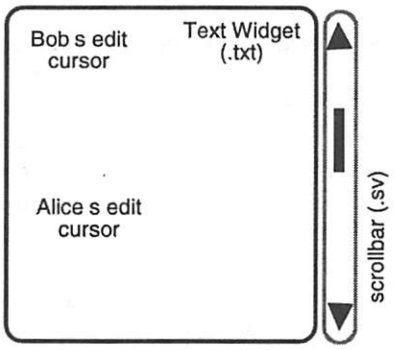
(b) Synchronized collaborative view using centralized server.



(c) Collaborative view using distributed architecture.



(d) Collaborative shared editor (for source code).



(e) Collaborative chat box (like talk window).

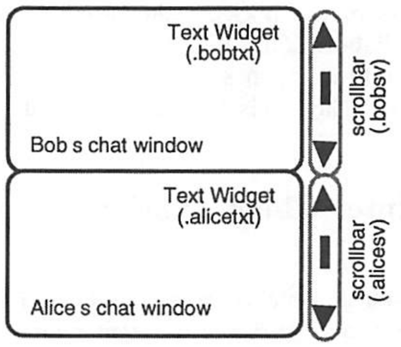


Fig. 1. Desirable collaborative configurations for a text widget with a scrollbar.

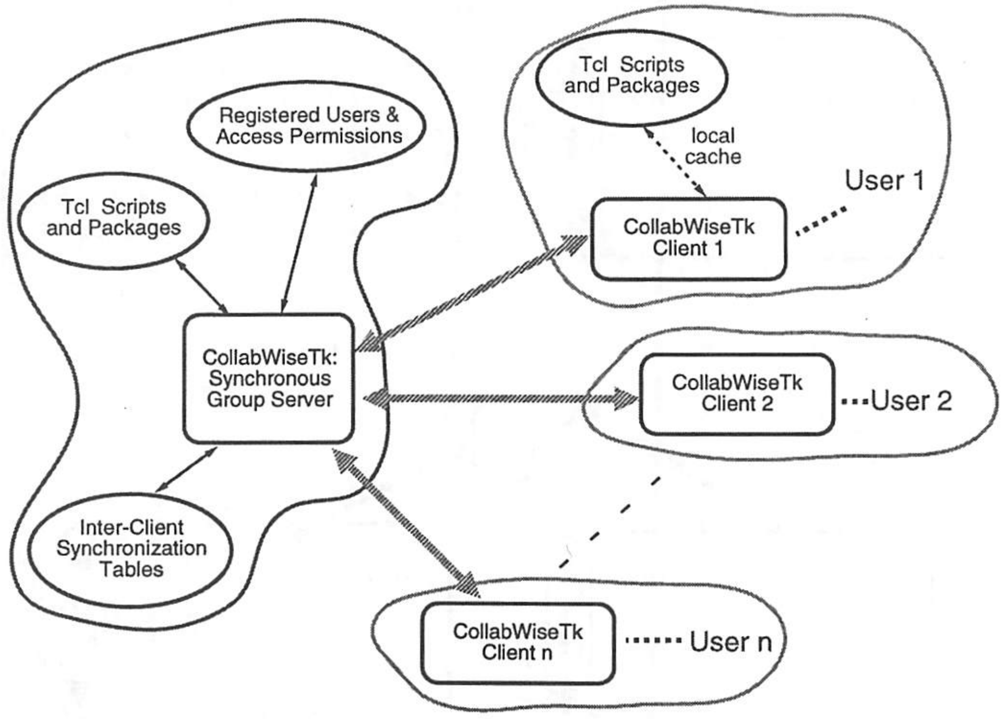


Fig. 2. A high level view of client/server architecture for collaboration.

with SGS and prompts the user to identify herself. Once the login process is completed, the user can access several different tcl applications, based on her access privileges, by invoking an appropriate configuration file from the inter-client synchronization table. Collaboration clients also maintain a local cache of the tcl applications for faster access.

The collaboration client also provides mechanisms to install new tcl scripts and packages in the group server repository. Privileged users can install such scripts and easily create configuration files on the server for others to access.

When two or more clients access the same configuration file of a tcl package, they are immediately set-up for collaboration. User-interactions with the tcl application are then sent to the group server, which in turn relays this information to all participating users.

4 Inter-Client Synchronization

Inter-client synchronizer allows the user to dynamically re-configure the different modes of collaboration for every primitive object contained within the tcl application GUI. A primitive object is an element of GUI with which a user can either interact or observe: (1) all tk widgets, such as button, label, entry, etc, are primitive objects; and (2) in addition, tagged items of a canvas and text are also considered as primitive objects.

Primitive objects can be configured into either one of the two states - *interact* or *observe*. When a primary object is configured to be in *observe* state, a user is prevented from interacting with it. For example, a user can be prevented from interacting with a text widget by removing all of its binding tags, as follows:

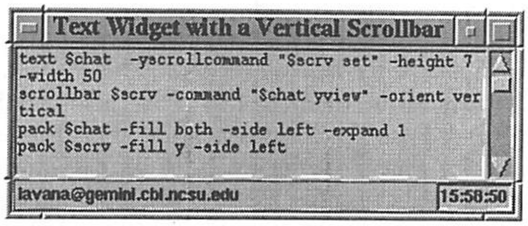
```
        bindtags .txt {""}
```

The same text widget can be brought back to an *interact* state merely by restoring its binding tags. A good introduction to *bindtags* can be found in [3].

When several users work collaboratively, each user invokes the tcl application locally. Therefore, all primitive objects of an application are replicated on each user machine. In addition, a user can now configure her primitive object to be in one of the two states, *interact* and *observe*, and link it to any of the participating users. This means that when there are two users, say Alice and Bob, each can configure their text widget to be in one of the four states:

- Interact/Alice,
- Observe/Alice,
- Interact/Bob, and
- Observe/Bob.

(a) Simple text widget with a vertical scrollbar.



Text Widget with a Vertical Scrollbar

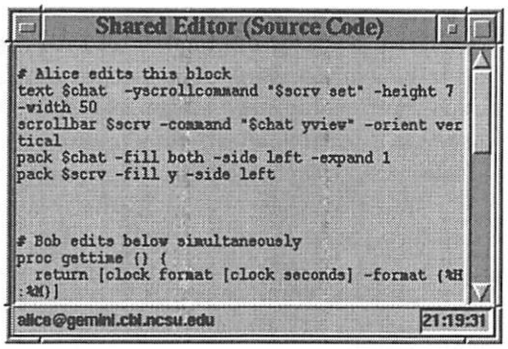
```
text $chat  -yscrollcommand "$scrv set" -height 7
-width 50
scrollbar $scrv -command "$chat yview" -orient ver
tical
pack $chat -fill both -side left -expand 1
pack $scrv -fill y -side left
```

lavana@gemini.cbl.ncsu.edu 15:58:50

(b) Widget tree structure

```
Widget tree
        |_ .
        |_ .f
        |    |_ .f.txt
        |    |_ .f.sv
        |_ .info
             |_ .info.user
             |_ .info.time
```

(c) Collaborative shared editor (for source code).

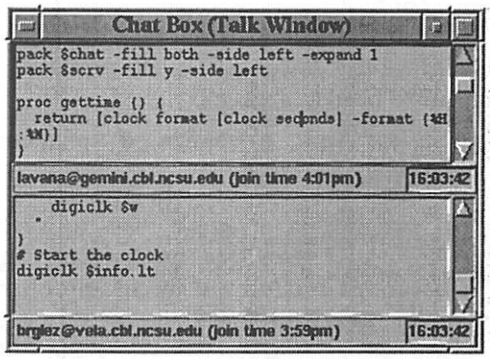


Shared Editor (Source Code)

```
# Alice edits this block
text $chat  -yscrollcommand "$scrv set" -height 7
-width 50
scrollbar $scrv -command "$chat yview" -orient ver
tical
pack $chat -fill both -side left -expand 1
pack $scrv -fill y -side left



# Bob edits below simultaneously
proc gettime {} {
  return [clock format [clock seconds] -format {%H
:%M}]
```

alice@gemini.cbl.ncsu.edu 21:19:31

(d) Collaborative chat box (like talk window).



Chat Box (Talk Window)

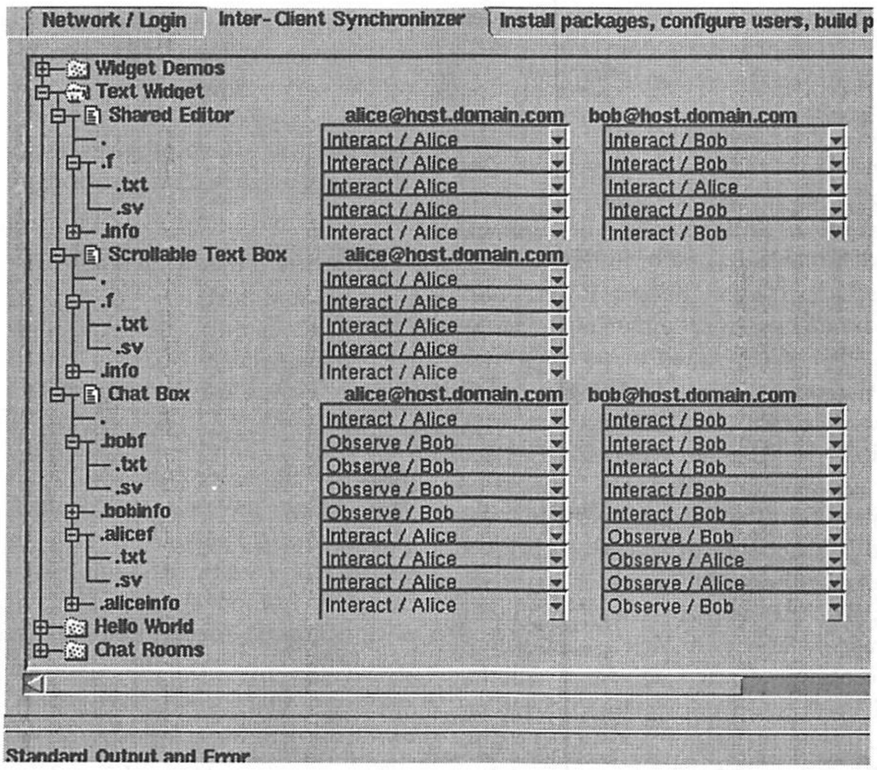
```
pack $chat -fill both -side left -expand 1
pack $scrv -fill y -side left

proc gettime {} {
  return [clock format [clock seconds] -format {%H
:%M}]
}
```

lavana@gemini.cbl.ncsu.edu (join time 4:01pm) 16:03:42

```
    digiclk $w
  .
}
# Start the clock
digiclk $info.lt
```

brglez@vela.cbl.ncsu.edu (join time 3:59pm) 16:03:42

(e) Inter-client synchronization table



Network / Login | Inter-Client Synchroninzer | Install packages, configure users, build p

Widget Demos
Text Widget

Shared Editor	alice@host.domain.com	bob@host.domain.com
.	Interact / Alice	Interact / Bob
.f	Interact / Alice	Interact / Bob
.txt	Interact / Alice	Interact / Alice
.sv	Interact / Alice	Interact / Bob
.info	Interact / Alice	Interact / Bob

Scrollable Text Box	alice@host.domain.com
.	Interact / Alice
.f	Interact / Alice
.txt	Interact / Alice
.sv	Interact / Alice
.info	Interact / Alice

Chat Box	alice@host.domain.com	bob@host.domain.com
.	Interact / Alice	Interact / Bob
.bobf	Observe / Bob	Interact / Bob
.txt	Observe / Bob	Interact / Bob
.sv	Observe / Bob	Interact / Bob
.bobinfo	Observe / Bob	Interact / Bob
.alicef	Interact / Alice	Observe / Bob
.txt	Interact / Alice	Observe / Alice
.sv	Interact / Alice	Observe / Alice
.aliceinfo	Interact / Alice	Observe / Bob

Hello World
Chat Rooms

Standard Output and Error

Fig. 3. Inter-client synchronization table used to configure a text widget into various collaborative modes.

We next explain the meaning of each of this state for Alice:

Interact/Alice: Alice is configured to interact with her own text widget. At the same time, Bob can observe her interactions, if he has configured his text widget to be in *Observe/Alice* state.

Observe/Alice: Alice is configured to observe her own text widget. This would result in activities on the text widget, unless Bob has configured himself to be in *Interact/Alice* state.

Interact/Bob: Alice is configured to interact with Bob's text widget. However, if Bob is also configured to be *Interact/Bob* state, then this could lead to potential conflicts.

Observe/Bob: Alice is configured to observe Bob's text widget.

The simple text widget example with a vertical scrollbar, shown in Figure 1(a), can be easily configured into various states by defining appropriate object state for the two widgets. Figure 3(a) shows a snapshot of a text widget GUI and Figure 3(b) shows its corresponding widget tree. This application is readily transformed, without any re-write, into: (1) collaborative shared editor for source codes (Figure 3c), and (2) a chat box window (Figure 3d), by providing an appropriate configuration file.

Figure 3(e) shows a snapshot of the inter-client synchronization table. It lists all the configuration files for the text widget in the left column, with their respective widget trees. Collaborative participants, if any, are listed in the adjacent columns. Thus, there are two participants for 'Shared Editor' and 'Chat Box', and only one participant for 'Scrollable Text Box'. The entries listed below each user corresponds to the current state of the respective widget. For example, the text widget '.txt' under 'Shared Editor' is listed as *Interact/Alice* for both the users, 'alice@host.domain.com' as well as 'bob@host.domain.com'. This implies that while Alice is interacting with her text widget, Bob also has the permission to simultaneously interact with Alice's text widget. The text widget is therefore shared among the two users thereby providing a means of real-time collaboration. Each such entry can be changed to a different state merely by clicking on the dropdown menus and selecting the desired state. We have used BWidget toolkit [12] to implement the GUI shown in Figure 3(e).

The examples shown above describe how each widget can be configured to suit one's requirements. However, as the size of the application grows, it can become very tedious for a user to configure each of this widget individually. Therefore we also provide a mechanism whereby if a user configures one widget to a specific state, then all of its subsequent children widgets also assume the same state. This becomes very useful when the user wants to start or stop interacting with the entire toplevel window and can be achieved by merely changing the toplevel window to the appropriate state.

5 CollabWiseTk Implementation

The client-server architecture of the `CollabWiseTk` toolkit is implemented using socket programming. Typically several clients may be connected to SGS at any time. In addition, clients may invoke several tcl applications and each tcl application may have several different collaborative participants. Figure 4 depicts one such scenario, where user1 on client1 has invoked TkAppln1, TkAppln3 and TkAppln4, user2 on client2 has invoked TkAppln2 and TkAppln4 and user3 on client3 has invoked TkAppln1 and TkAppln2. An application that is common among users imply that those users are its collaborative participants. For example, TkAppln1 is rendered collaborative among user1 and user3. This is also shown as a dashed line linking TkAppln1 to client1 and client3 on the server side.

The following table provides such a relationship for the example shown in Figure 4:

Application/Username	User1	User2	User3
TkAppln1	*		*
TkAppln2		*	*
TkAppln3	*		
TkAppln4	*	*	

As the number of participants and the number of applications increase, this could lead to very complicated relationships. Also, it is important that the server as well as the client maintain this information in a clean fashion without mixing up any information with one another. We, therefore, invoke a new interpreter for each client that connects to the server. In addition, a new interpreter is also created for every application that is invoked by all the clients. Similarly, the client also invokes a new interpretor for every application that the user invokes. These interpreters are invoked using the command `interp create -safe` – we make use of *safe* interpreters to provide adequate security among the client/server

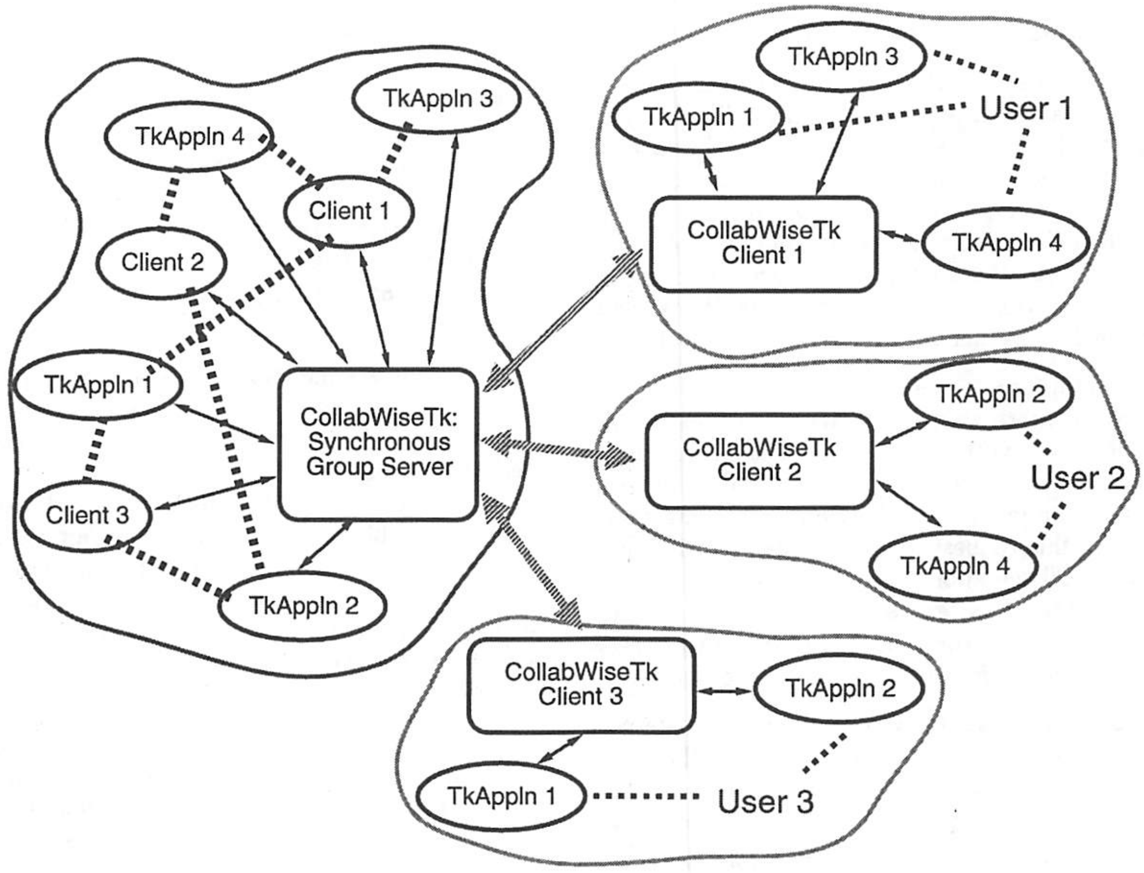


Fig. 4. Dependency relationships for a three-participant collaborative session in progress.

communication.

We next describe the mechanism used to capture events corresponding to user interactions and how these events are broadcast to its collaborative participants.

Interpreters on SGC are configured to provide a set of Tk commands such that in addition to being able to create, configure and delete generic widgets, these also include functionality to trap all the user interactions with the widget. Once the user interactions are trapped by the interpreter, it is upto sole discretion of the application-specific interpreter on how to process this information and it depends on the state of the inter-client synchronization table. For example, once the double-click on a button of `TkAppln 1` by `User 1` is captured, the interpreter does the following:

1. Check the current state of the button in inter-client synchronization table;

2. If the current state of the button happens to be in observe mode, then it immediately stops processing the event any further;

3. If the current state of the button happens to be in interact mode, then it processes the corresponding event;

4. If any of the collaborative participants are setup in observe mode for this button, then the interpreter transmits the event to the SGS server for further processing.

SGC interpreters also perform the functionality to receive and process events that are transmitted by collaborative participants.

Client-specific interpreters on SGS are configured to:

1. receive events from their respective clients and forward it to the application-specific interpreter for broadcasting it to other relevant clients, and

2. relay or send the processed event from application-specific interpreters to their respective clients.

On the other hand, application-specific interpreters on SGS actually perform a inter-client synchronization table lookup to process an event and send it to all the clients that are in observe state.

Since users are allowed to choose to choose and configure the state of a widget in an application, it is very easy for widgets to fall out of synchronization with corresponding widgets of collaborating participants. For example, this occurs when Alice changes the state of her text widget from `Interact/Alice` to `Observe/Bob`. Before Alice can start observing the Bob's interactions with his text widget, Alice has to first synchronize her text widget with that of Bob.

The client interpreter of Alice sends a request to SGS to retrieve information necessary for her text widget to synchronize with that of Bob. The SGS server in turn transmits the request to Bob's client interpreter and waits for an answer. The Bob's client interpreter processes this request and responds accordingly by providing all the appropriate information about the current state of its text widget. Once, the two text widgets are synchronized, Alice goes into observe mode and starts following Bob's interactions.

SGS and the collaborative client communicate with one another using a well-defined set of APIs. There are basically two types of API commands:

- synchronous API commands, which block the sequence of execution by waiting for the results; and
- asynchronous API commands, which do not block and wait for the results of the execution.

When client1 sends a user interaction to a collaborative participant client2, it is not necessary for the client1 to verify whether client2 has managed to receive this information or not. Hence, this falls into asynchronous type of command.

On the other hand, when client1 requests specific information about client2, such as the status of its scrollbar, then it is necessary to issue a synchronous command. However, this implies that synchronous commands will block the socket channel between the client-server and prevent communication of other commands. Therefore, we always send asynchronous commands over the socket channel, and make use of the 'vwait' command to implement the synchronous command.

Accordingly, we have developed two simple procedures to handle these two types of commands, as shown below:

```
##
# 1. Asynchronous transmission
#    sid = socket id
proc sendNforget {sid cmd} {
  transmitSocket $sid {} $cmd
};# End of proc sendNforget

##
# 2. Synchronous transmission
proc sendNreceive {sid cmd} {
  set returnId [clock clicks]
  transmitSocket $sid $returnId $cmd
  vwait $::returnId
};# End of proc sendNreceive
```

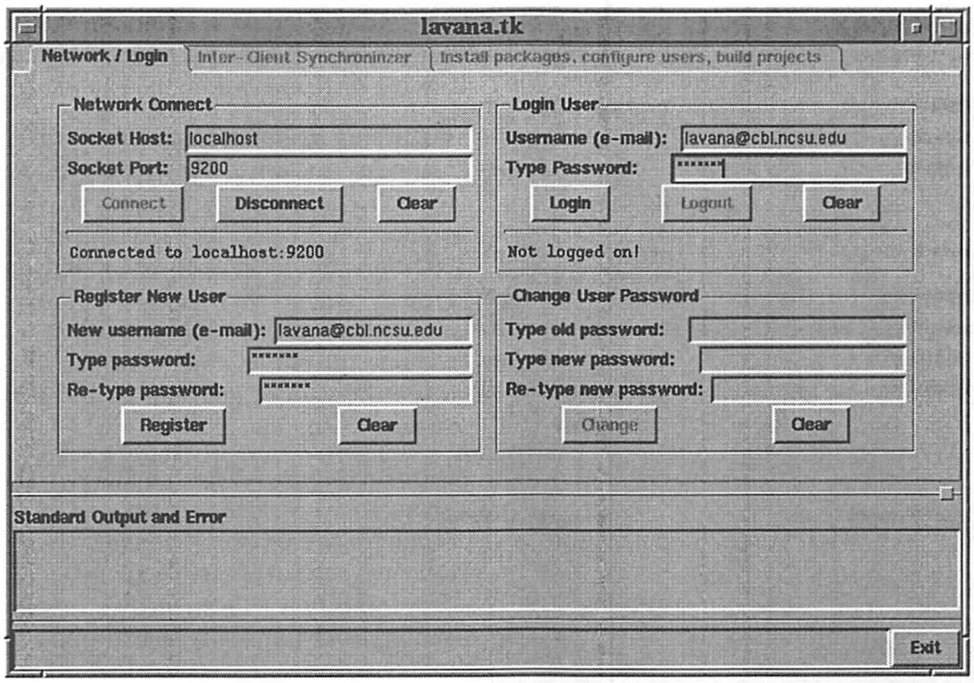
The procedure `sendNforget`, as the name suggests, sends the command across the socket channel and returns immediately. On the other hand, the procedure `sendNreceive` transmits a unique variable name, generated using the clock command and waits for this variable to be set using the vwait command. When this data is received on the other side, the presence of a variable name implies that a result is expected and therefore it transmits back the results such that this unique variable name is set to contain the results.

The notification of user interactions with a widget is achieved by analyzing the entire set of commands in tk widgets and re-defining these commands appropriately. Specifically, the original text widget is renamed and a new procedure is created by the same name. This new procedure performs actions that are necessary to:

- inform the group server, whenever a new text widget is created and if some other client is observing this text widget;
- create a new procedure for the text widget pathname that selectively sends similar information to group server.

The following tcl code snippet shows a simplified example of how a text widget can be made collaborative.

```
## Rename text widget
#
rename text text-Org
proc text {pathname args} {
  sendNforget $::sid "text $pathname $args"
  set ret [uplevel 1 text-Org $pathname $args]
  #rename $pathname cmd which just got created
  rename $pathname $pathname-Org
  proc $pathname {args} {
    # process $args
    switch -- $opt {
      cget {
        # send nothing to remote server, OR
        # use "sendNreceive $pathname cget"
```



Network Connect
Socket Host: localhost
Socket Port: 9200
Connected to localhost:9200

Login User
Username (e-mail): lavana@cbl.ncsu.edu
Type Password: *******
Not logged on!

Register New User
New username (e-mail): lavana@cbl.ncsu.edu
Type password: *******
Re-type password: *******

Change User Password
Type old password:
Type new password:
Re-type new password:

Standard Output and Error

Fig. 5. Startup window of a CollabWiseTk client.

```
        # cmd to get option value from rmt
        # user, when configured for
        # interaction with rmt user.
      }
      insert {
       sendNforget $::sid \
             "$pathname insert char"
      }
    };# End of switch stmt
    # ....
  };# End of proc $pathname
  return $ret
};# End of proc text
```

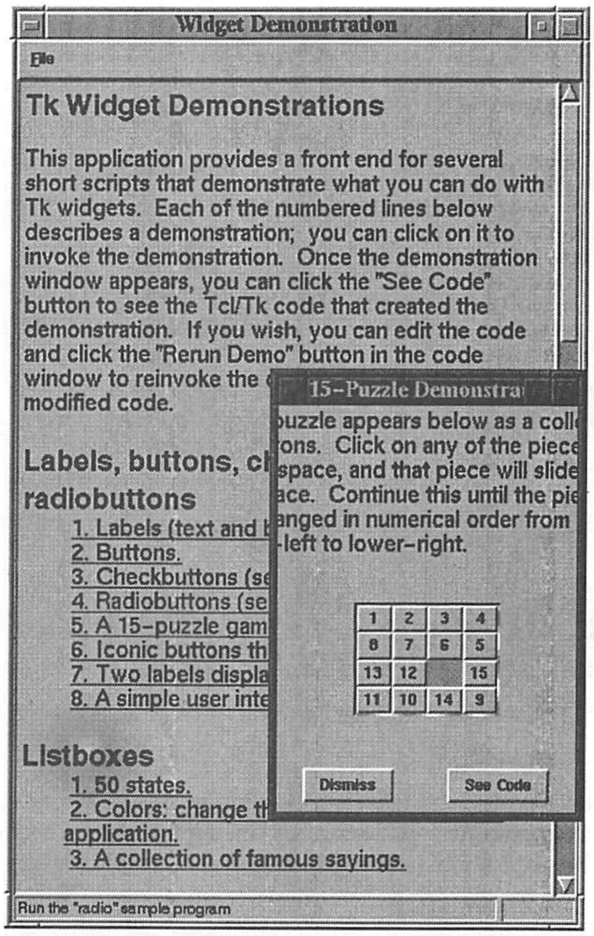
Whenever the application invokes a text widget, the above procedure is called. It first informs SGS about the creation of the text widget using the non-blocking command `sendNforget`. It then proceeds with the creation of the actual text widget. Since creation of a text also creates a new command by its widget name called `pathname`, it re-defines `pathname` to `pathname-org` and creates a wrapper script for it. Other Tk widgets are implemented similarly.

6 Testbed and Experiments

We decided to use the Tk widget demonstrations, distributed with the core Tcl/Tk, as a test-bed for testing the `CollabWiseTk` toolkit. We have chosen these demos because they not only cover most of the commands in the tk widget set, but also demonstrate usefulness of the toolkit in rendering these applications collaborative. The experiments need to be conducted in two phases:

- manually invoke all the demos and verify their operation in several different configuration modes; and
- setup a testbed and conduct a series of experiments to evaluate the performance and scalability of this architecture.

Our current implementation of the toolkit allows us to collaborate many of the listed demos. However, we have not yet implemented: (1) one major tk widget, namely the canvas widget; and (2) the tagged items on the text/canvas widget in our collaboration toolkit. Figure 5 shows the initial startup window of a collaborative client, which allows the user to connect and login to the synchronous group server. Figure 6 shows the main window of the tk widget demonstrations invoked in collaborative mode. Specifically, it shows a 15-puzzle game which has been made collaborative for two players as described next. First player can only click on the odd buttons whereas the second player can only click on the even buttons. Two kinds of games can be played with such a configuration: (1) players assists one another in completing the game at the earliest; or (2) one player tries to prevent the other player from completing the game.



Tk Widget Demonstrations

This application provides a front end for several short scripts that demonstrate what you can do with Tk widgets. Each of the numbered lines below describes a demonstration; you can click on it to invoke the demonstration. Once the demonstration window appears, you can click the "See Code" button to see the Tcl/Tk code that created the demonstration. If you wish, you can edit the code and click the "Rerun Demo" button in the code window to reinvoke the
modified code.

Labels, buttons, c
radiobuttons
1. Labels (text and
2. Buttons.
3. Checkbuttons (s
4. Radiobuttons (se
5. A 15-puzzle gam
6. Iconic buttons th
7. Two labels displa
8. A simple user inte

Listboxes
1. 50 states.
2. Colors: change th
application.
3. A collection of famous sayings.

Run the "radio" sample program

puzzle appears below as a coll
ons. Click on any of the piec
space, and that piece will slid
ace. Continue this until the pi
anged in numerical order from
-left to lower-right.

Fig. 6. Collaborative Tk widget demos.

7 Software Evaluation

We evaluate the *CollabWiseTk* toolkit in the context of several factors below.

User configurability. We provide the flexibility to save the mode of collaboration for a specific application in a static file. A programmer can thus anticipate a variety of useful collaboration modes and save it in separate configuration files. Users can then invoke the application with appropriate configuration file during collaborative sessions. However, users are not limited to using the collaborative mode defined in these files, but have the flexibility to also change the mode of collaboration, during run-time, to suit their specific needs.

System architecture. We have chosen a hybrid architecture to implement CollabWiseTk. A replica of the single-user application executes on every user's machine thereby providing good response times for local interactions. On the other hand, a centralized server is used for synchronization and maintaining consistent state information. This may result in performance bottlenecks where high interactivity is needed for increased number of participants.

Group awareness. Inter-client synchronization table maintains a list of active sessions corresponding to each application being shared. Additionally, every session not only lists the number of users actively collaborating on an application, but also displays the type of user activity such as interact or observe for a specific widget. This enables other users to determine the status of a particular user in respect to a specific widget and aids in increasing the group awareness significantly.

Floor control. The toolkit provides a very fine granularity over floor-control and allows the users to configure the interaction mode down to the widget element, instead of merely allowing control of the entire application. Additionally, users also have the capability to dynamically change and allow other users to interact with a specific widget of choice whenever the need arises. This provides better flexibility in letting users drive the mode of collaboration to suit their needs, rather than a programmer trying to anticipate all the collaborative needs for design specific collaborative-aware application.

Scalability. The scalability of the toolkit depends on the users and how they decide to share a specific session during runtime. If users want to be aware of all the activities of the other users, then there will naturally be a performance hit when the number of users increase. However, it is expected that most users will not work in such fully shared mode and would prefer sharing only part of the widgets from the entire application. This would minimize the communication overhead and hence this toolkit would offer good scalability even when the number of users increase.

Limitations. The architecture relies on a single centralized server to share the information for collaboration. This can result in disruption of collaboration services if the server fails for any reason.

8 Software Availability and Status

The `CollabWiseTk` toolkit has been currently implemented for most of the tk widgets, except canvas. We also plan to make the toolkit available on the Web, once the canvas widget is fully implemented and we go through in-house testing phase.

Further details about the current status of these packages will be made available under:
`http://www.cbl.ncsu.edu/software`

9 Conclusions

We have demonstrated the capabilities of the `CollabWiseTk` toolkit by readily transforming most of the existing Tk Widget Demos into collaborative applications. Furthermore, the versatility of the toolkit is realized by the flexibility that it provides in rendering a stand-alone application into a variety of collaborative modes with little amount of work. Additionally, the functionality of these collaborative clients can be dynamically re-configured by the participants - thereby making it a very useful toolkit.

The re-configurability of the low-level primitives can be combined with one another to form several useful mega-widgets that better reflect users's models of work. This is illustrated with the help of a text widget and a vertical scrollbar, which is rendered collaborative as

- a shared editor,
- a chat box window, or
- a local text box supporting remote edits.

Similarly, other primitive widgets can be combined to create numerous useful applications.

References

[1] F. Brglez, H. Lavana, Z. Fu, D. Ghosh, L. I. Moffitt, S. Nelson, J. M. Smith, and J. Zhou. Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience, February 2000. Seventh Annual Tcl/Tk Conference, Feb 14-18, 2000, Austin, Texas.

[2] F. Brglez. Frontiers of Collaborative Computing on the Internet, A Graduate Course Experiment, January 1999. Two project reports, published after the completion of the course, are also available from the course home page under `http://www.cbl.ncsu.edu/~brglez/csc591b/`.

[3] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming.* Addison-Wesley, 1998.

[4] GroupKit Version 5.1. Published under URL `http://www.cpsc.ucalgary.ca/grouplab/groupkit`, 1998.

[5] TANGO: Collaboratory for the Web. Published under URL `http://trurl.npac.syr.edu/tango`, 1998.

[6] G. Konduri and A. Chandrakasan. A Framework for Collaborative and Distributed Web-Based Design. In *Proceedings of the 36th Design Automation Conference*, June 1999.

[7] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proceedings of the 34th Design Automation Conference*, pages 553–558, June 1997. Also available at `http://www.cbl.ncsu.edu/publications/-#1997-DAC-Lavana`.

[8] H. Lavana, A. Khetawat, and F. Brglez. Internet-based Workflows: A Paradigm for Dynamically Reconfigurable Desktop Environments. In *ACM Proceedings of the International Conference on Supporting Group Work*, Nov 1997. Also available at `http://www.cbl.ncsu.edu/-publications/#1997-GROUP-Lavana`.

[9] A. Khetawat, H. Lavana, and F. Brglez. Internet-based Desktops in Tcl/Tk: Collaborative and Recordable. In *Sixth Annual Tcl/Tk Conference.* USENIX, September 1998. Also available at `http://www.cbl.ncsu.edu/-publications/#1998-TclTk-Khetawat`.

[10] S. Greenberg and M. Roseman. Groupware Toolkits for Synchronous Work. In M. Beaoudouin-Lafon, editor, *Computer-Supported Cooperative Work, Trends in Software Series.* John Wiley & Sons Ltd., 1998. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under http://www.cpsc.ucalgary.ca/projects/grouplab-/papers/1998/98-GroupwareToolkits.Wiley/Report96-589-09/report96-589-09.pdf.

[11] S. Greenberg. Real Time Distributed Collaboration. In P. Dasgupta and J. E. Urban, editor, *Encyclopedia of Distributed Computing.* Kluwer Academic Publishers, 1999. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under http://www.cpsc.ucalgary.ca/-projects/grouplab/papers/1998/98-Encyclopedia-Distrib/encyclopedia-realtime-collaboration.pdf.

[12] BWidget Toolkit. Published under URL `http://www.unifix-online.com/BWidget`, 1999.

GDBTk - Integrating Tcl/Tk into a recalcitrant command-line application.

Jim Ingham
Cygnus Solutions
jingham@cygnus.com

Abstract:

This paper will describe the some of the lessons we learned embedding Tcl/Tk into the Gnu Debugger (GDB). GDB is a command-line application, and there were a number of problems which this caused, particularly when using Tk within the application. We will describe a few of these problems, and the solutions we came up with. The paper will also give a brief overview of GDBTk itself.

Introduction

GDBTk started as a skunkworks project by Stu Grossman in 1994. There had been a number of other GUI interfaces to gdb before this (xgdb, xxgdb, the Emacs interface, and later DDD). GDBTk was distinguished from them by having the GUI actually linked with the debugger. Most of the other GUI's for GDB spawned GDB as a child process and then used the GDB command language to drive GDB, and parsed the output to get information back out of it. This method overcomes several significant problems with living in the same application as GDB, but has a number of drawbacks.

First among the drawbacks, there are some operations which are too slow when the information has to be printed by GDB, and then parsed in a separate program. Examples of this are presenting a variable display when there are many complicated structures in scope, or performing disassembly (or mixed source/assembly display) on large functions. Second, GDB has a lot of knowledge about the executable which it does not necessarily make available through the command line, or does not make available in a compact form. Efficient listing of backtraces, or function lookup are several examples of this. Thirdly, and perhaps most importantly, having a real scripting language embedded into GDB has persuasive advantages all on its own that fully justify the effort.

In the first section of this paper, I will describe the general structure of GDBTk. In the second, I will lay out a few of the problems we had in working with GDB, and the solutions we came up with, hopefully drawing some lessons for others faced with a similar task. Then in the last section, I will give a brief tour of the current state of GDBTk, and point out some of the places where the implementation in Tcl has caused particular problems, as well as some instances where the use of Tcl really shines

Since the initial version by Stu Grossman, there have been two other GDB GUI's produced at Cygnus. Before Tk was available on Windows, Steve Chamberlin wrote the (ill-fated) WinGDB, an MFC based Windows application. However, as soon as a cross-platform version of Tcl/Tk was available, development on this was stopped, and Martin Hunt, Keith Seitz, Tom Tromey and Ian Lance Taylor developed a much enhanced version of the original GDBTk code. I joined Cygnus to work on this project after the Sun Tcl effort folded in March 1998. Since that time GDBTk has seen considerable improvement, and is also included as the debugger interface in the CodeFusion IDE[1] from Cygnus. In August of 1999, GDBTk (now called Insight) was released under GPL to the net[2].

GDBTk Internals:

GDBTk is constructed in four major parts. The first is GDB itself. Most of the core functionality of GDBTk resides in GDB. We try not to duplicate code that is handled in the GDB core wherever possible, though, as I will mention below, there are cases where this is unavoidable. Also, although, due to the strictures of the FSF which owns the actual copyrights to GDB, we cannot actually introduce Tcl code into the GDB core, GDB itself contains a number of hooks - callouts to client-defined functions - that GDBTk uses to find out about interesting events in the debugger.

The second part of GDBTk is its implementation of the hook functions. These hooks fall into two broad categories. One type provides the GUI with notification of events that happen outside of its control in GDB. An example of this is that, when GDB hits a temporary breakpoint, it deletes the breakpoint. The GUI needs to be notified of this fact, so that it can remove the breakpoint from the breakpoint window. The other main function of the hooks is to allow GDBTk to keep in synch with the standard GDB command language console. Anything significant that can be changed in the console is given a hook function, and that is how the GUI is notified of the changes the user makes there.

The third part of GDBTk's implementation is a Tcl extension that provides access to many of GDB's internal API's. On the simplest level, we have the `gdb_cmd` command, which just passes its argument to the GDB command parser, and returns whatever the GDB command would have written to stdout. At the same time, we introduce a command: `tk` into the GDB command interpreter, so we can run Tcl code from the GDB console window.

When you want to instruct GDB to perform some action - set or delete a breakpoint, open an executable file, detach from a running program, gdb_cmd is all you need. The output is either very simple or non-existent, so duplicating the functionality in Tcl would serve no purpose.

However, for commands that get information out of GDB - getting variable values, or reading symbols out of the symbol table, there is either no GDB equivalent, or the equivalent has output which is in an inconvenient form. In these cases, we have added Tcl commands that perform the tasks. Some examples of given below:

```
gdb_listfiles..............Lists the files in the current
   executable.

gdb_listfuncs file.........List the functions in "file"

gdb_loc symbol.............Returns a complete specifica-
   tion (file, line number and pc) for the symbol "sym-
   bol".

gdb_get_regs...............Returns a list containing the
   current register values.
```

There are 49 commands in the gdbtk command set.

1. See http://www.cygnus.com/codefusion
2. See http://.sourceware.cygnus.com/insight

The most interesting set of C-based Tcl commands in GDBTk are the ones that deal with listing the variables in the current scope. This was an area where parsing the GDB output was difficult and slow. Keith Seitz created a set of Tcl commands that return the blocks in a function, the variables in each block and then a Tk-like set of commands that create Tcl access commands for each variable, or indeed for any valid expression. The access commands allow you to access the variable in the executable - query and change its value, get its type, dereference pointers, and get all the children of a structure or all the instance data of a C++ object. Using these, we are able to construct a local variables window which can update itself with no visible lag on any reasonable system, even in test cases with 600 variables in the current scope.

The final component of GDBTk is the actual GUI implementation code. We chose IncrTcl for the implementation, since this we knew this was going to be a big project, and we felt this was the best way to manage the complexity. Since much of the code was written originally for Itcl1.5, we were much less ambitious about the class hierarchies than we could be if we were writing it with Itcl3.0. We have a general ManagedWin base class that all the windows derive from. This allows us to coordinate the windows, and is a convenient factory for the windows that we want to be unique. We also have generic dialog classes, etc. And each window type is a separate class. Needless to say, we are very happy with IncrTcl, it has made the project much more maintainable.

We also use a number of other components from the Net: a somewhat hacked version of Brian Oakely's ComboBox[3], a number of the IncrWidgets[4], and TkTable[5]. We also use Tix, though given that it is currently maintained, and is showing its age, we have been backing out all the uses of it that we can. The only GUI element that we can currently find only in Tix is a Tree-driven table widget. The TkTable does well for things like the memory display. The BLT[6] her widget makes a good tree. But for the variable window, we need both the hierarchical Tree structure to represent structures, and a table to list variable types and values in columnar form. The value elements also need to be editable. We tried the experiment of coupling BLT's hier widget with the TkTable, but had no success in getting the two to expand in synch. The visual artifacts were very distracting.

What about GDB makes this project difficult?

GDB was designed to be a command line application, and was never intended either as a callable library, or as a scriptable application. The difficulties this causes fall into two main categories: lack of a real "callable interface", and the blocking behavior typical of a command line application. I will next describe these features in turn:

1.) Lack of a "Callable Interface": While GDB does have a command language, for the most part that command language lacks the notion of "return values" for command operations. Moreover, since it does not itself ever use things like the values of structures internally, it has no facility to return this information at the C level. Instead, for instance, when you want the value of a variable or expression, you call a routine that evaluates the expression, and in the course of the evaluation, prints out the type and value of the variable. If the variable is a complicated object, like a structure, the subelements and their types are recursively printed as well. The same is true for other useful bits of information like accessing the registers, the stack, and the symbols that the debugger knows about. Needless to say, this makes it very hard to write a reasonable set of Tcl commands that access the information you need to get out of GDB.

2.) Blocking Behavior: GDB spends a lot of its time waiting for the debugger target to do something interesting, whether this be hit a breakpoint, evaluate an inferior function call, or return some requested bit of data. During this time, there is not much useful that you can do with the debugger. Because of this, the original designers did not make any effort not to block while waiting. After all, if you did need to wake up the debugger, you could

3. See http://www1.clearlight.com/~oakley/tcl/combobox/index.html
4. See http://www.tcltk.com/iwidgets
5. http://www.purl.org/net/hobbs/tcl/capp/tkTable/tkTable.html
6. See http://www.tcltk.com/blt

always send Control-C, which would interrupt the execution and return control to the user. This is very bad for a GUI application, however, since the UI has to deal with things like repainting while the target is running.

One other minor annoyance that is worth noting is the exception handling in GDB. GDB was designed to handle all errors by setjmp/longjmp. Essentially, the program does a setjmp in the command loop at the top of the application, and if it ever runs into trouble, it just longjmps back to the top level. There are, of course, facilities to register "clean-ups" to deallocate memory etc. This mode of exception handling is very simple to implement, since you don't have to bother with returning status, and figuring out how to back out nicely, which is why it was chosen for GDB. However, it is fatal for an system like Tcl which keeps vital information on the C-stack.

The solution to this problem in GDBTk is to use a generic "call wrapper" as the command proc for all the C based Tcl commands. The actual command is passed in the ClientData field. The call wrapper stores away the old jump buffer, does some other housekeeping, does its own setjmp, and finally calls the Tcl command. This works fairly well, but in some cases is too coarse grained to allow the Tcl command to recover properly. We keep a steadily growing set of wrapped versions of useful gdb calls, and use those when it matters.

Solutions -

Callable Interface:

This is perhaps the most serious problem with GDB. The progressive accumulation of results is endemic throughout the program, and in many cases there is no other interface to the data, so you would have to start from scratch and rewrite the functionality. Doing this is complicated by the fact that many people - including DDD and the other external debuggers - depend on the exact form of the output from GDB, so you would also have to exactly replicate the current functionality, or leave both implementations in place, which is a code-maintenance nightmare. There were simply not the resources for this level of rewrite, so a different approach had to be found.

Solution 1: *puts hooks*

The first solution - used in the original GDBTk - was to search for all occurrences of the C library stdout routines, and replace them with a gdb functions that ultimately route through a single function - fputs_unfiltered. This function would just call fputs in the non-GUI case, and would call a function in Tcl-land which would accumulate the characters into the Tcl result, in the GUI case. This was natural for gdb, because it already had code to handle paging of output, so most of the printing was already routed through gdb functions.

This method, while it can be made to work, and is in fact the current base for GDBTk, has several inherent flaws. One has to do with mixing streams of output. An obvious example is that you have to strictly separate the error and output streams, or you will mix error strings into the output you are trying to parse, often causing the parsing to fail altogether, but at least presenting erroneous data in the GUI.

In some cases, this is an easily solvable problem, of course, simply use separate streams for output & error. However, this does not help you in cases where, in the course of gathering one bit of information you have to call into a function which outputs some other information. Ring all the changes on this - only SOMETIMES calls into... - and it leads to a fragile framework.

The next flaw in this method is that it forces you to parse string data. Particularly when getting the values of structures, this parsing task can be quite complicated and error prone. We have alleviated this somewhat by introducing a flag that we pass to the accumulator hook to tell it to add each new element to the result as a Tcl list element. This works sometimes, but often gdb does not accumulate the results in a particularly coherent way. Moreover, the output is not self-descriptive, so as the data is output, information about what it is lost, and has to be reconstituted in the Tcl code, once again leading to errors.

Solution 2: *Libgdb*

To solve these problems, Cygnus started to work on the "libgdb" project. This is a modest project, we are not rewriting GDB to be a true callable interface. The goal is rather to revise its output capabilities so that data can be annotated as it is produced, and so that output streams can be separated. There were two other restrictions to the design of the libgdb project, it has to replicate exactly the current gdb output, so we don't break extant scripts and clients of the command line, and it has to be adaptable to any scripting interface, since GDB is officially an FSF project, and Tcl is persona non grata in the eyes of the FSF.

The way it works is that we introduce a function table that contains another set of calls to do the printing. This as a richer set of calls than the earlier GDB set, and allow you to qualify the output as you print it. In the core gdb code, the calls appear as ui_out_..., but this is actually a #define that calls into the function table of the currently installed output builder. So for instance, the old version of printing breakpoint information looked like (printf_filtered is one of the original gdb printing functions):

```
printf_filtered ("Breakpoint %d", b->number);

if (b->source_file)
  printf_filtered ("at : file %s, line %d.",
                   b->source_file, b->line_number);
```

This is translated in the libgdb mode to (uiout is the ui_out wrapper for a stream):

```
ui_out_text (uiout, "Breakpoint ");
ui_out_list_begin (uiout, "bkpt");
ui_out_field_int (uiout, "number", b->number);
if (b->source_file
  {
    ui_out_text (uiout, ": file ");
    ui_out_field_string (uiout, "file",
                         b->source_file);
    ui_out_text (uiout, ", line ");
    ui_out_field_int (uiout, "line", b->line_number);
    ui_out_text (uiout, ".");
  }
  ui_out_list_end (uiout);
```

This example shows several elements of the libgdb interface. `ui_out_text` is a routine that outputs "human readability" data - a scripting interface should ignore this. This allows us to exactly reproduce the current gdb output, while not burdening the parser on the other end. libgdb also has the notion of keyed lists, much like the TclX keyed list facility. The lists themselves are also labeled. In this example `ui_out_list_begin` begins the accumulation of the list called bpkt, and the `ui_out_field_*` calls add fields to the list. This makes the output self-descriptive, which in turn makes the clients robust, since they are no longer dependent on output order.

There are a number of other nice features in the Libgdb interface. It supports building tables. It also has the notion of named streams, so you can accumulate to a named stream, and if some other output is generated in some function that you have called, it will go to the default output, so it won't get lost, but it will not pollute the result you are currently accumulating. The desirable property of this solution, for GDB, is that the code does not need to be inverted in order to use it from Tcl. All the changes are local, and can be made with high confidence. Given the complexity of GDB, as well as the age of the code, this is a very desirable property.

This code is not yet finalized, and except in a sandbox has not been incorporated into GDBTk yet. This will happen over the next six months, maybe even by the time you read this. We will also use this work to build the generic scripting interface to GDB, and we will implement a Tcl instantiation, while the official FSF version will probably use Guile.

Blocking Behavior:

This problem critically effects the look and feel of the GDBTk. It is the reason why almost all the other GDB GUI's chose to run in a separate process. It is important that the UI repaint itself, and that it continue to be interruptible, no matter what the target is doing. So you need to always be listening to events coming from the connection to the X-Server.

Solution 1: *Timers*

Our first solution to the problem was to use signals to wake up the GUI while gdb was blocked. There are a number of ways you can do this. One is to use SIGIO on the connection to the X Server. However, SIGIO is not delivered at all on Linux, and not reliably on other systems. So in the end we had to resort to using timers. Before each call into GDB that we knew was going to block, we would start up a timer, and then refresh the GUI in the SIGALRM handler.

This method requires some care, since you intend to run Tcl code in the SIGALRM handler. Remember, the point is JUST to refresh the interface, and not to interrupt the call that is currently blocking. You need to make sure that you never get the SIGALRM when you are not blocking, since this will lead to random crashes. This means that the time-out has to be fairly coarse-grained, and started and stopped fairly close to the place where you will block.

Although this is the method that GDBTk currently employs, it is highly unsatisfactory. It is too easy to miss a place where gdb can block. This is particularly a problem when something unexpected happens, like you lose connection to a remote board, and then a query that normally takes no time, and occurs too far down the call chain for the timers to be conveniently set and unset, suddenly blocks for the length of the "connection lost" timeout, which is generally ~5 seconds. It also causes odd occasional corruption if the timer happens to fire just on the way out of a call. The lesson we learned from this is that while with a lot of work, you can manage to fake an essentially blocking application to look like an event driven one, in the long run you will lose too often to make the fake convincing.

Solution 2: *A Real Asynchronous Event Loop*

Ultimately, we decided that there was no way to get GDB to behave well when driven by a GUI, unless it really was a true event driven application. Furthermore, there is quite a bit of information that it is legitimate to access while the target is running. You might want to browse the breakpoint list, or list files, or search for functions in the executable. So there was a good deal of motivation to make this switch, even though it involved reworking a large chunk of gdb.

At this point there were two models that we could follow. One was to make gdb a multi-threaded application, and the other was to treat the inputs that GDB had as event sources, and write a select-based event loop on the Tcl model.

Most of the inputs to gdb are, in fact, standard Unix event sources. Gdb can talk to remote targets either through the serial port, or over TCP/IP. The most common UNIX debugging interface is the procfs file system which is selectable. The other common UNIX debugging interface is ptrace. In this case, the calls can be made in a non-blocking mode, and then you get a SIGCHLD when the inferior process returns control to the debugger.

All this fit quite naturally in the Tcl event model. This is a well tested model, and we were confident that we could handle all the many platforms GDB runs on with this solution, whereas we had less confidence that we could get a multi-threaded application to run will cross all of GDB's supported platforms. The actual event handling code in GDB is strongly influenced by the Tcl model. We could not, of course, just adopt the Tcl code for political reasons, but we used the Tcl 8.0 notifier as the basis for the new design. It was also constructed so that we could easily convert all the input channels to Tcl channels, and use the Tcl event loop for GDBTk.

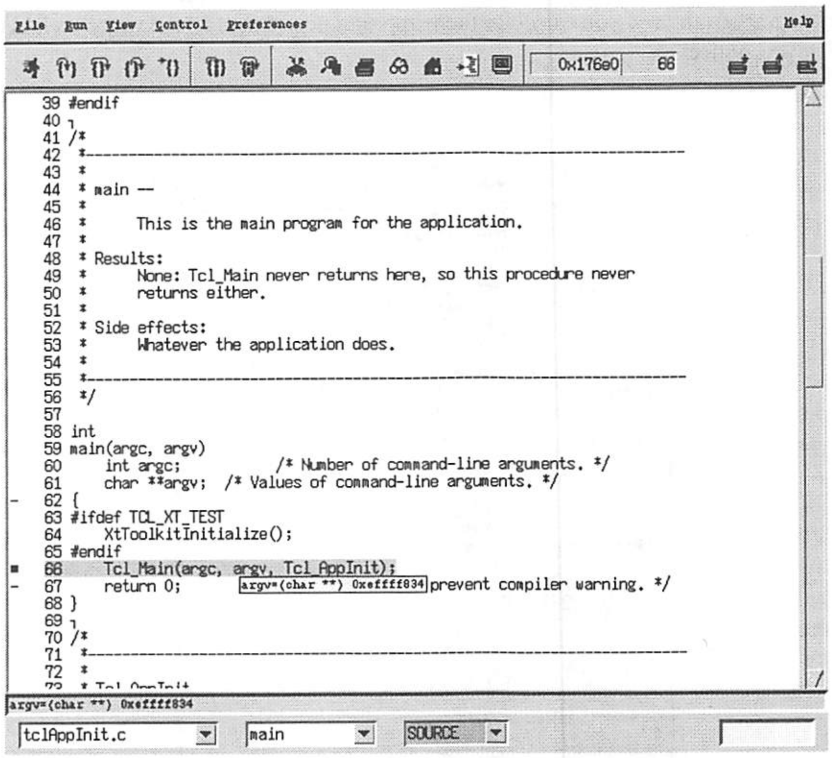
This was quite a major job, since GDB like Tcl keeps a much of its own state information on the stack. Interactions with the debugee can be quite complicated, often involving many restarts in what seems to the user (and to the higher level GDB API's) like an atomic operation. The work is not entirely complete, but by the time this paper is published all the code will be in place, and GDBTk will use it.

Brief Tour of GDBTk:

The rest of this paper consists of a short tour of some of the main screens in GDBTk. In general, there were no UI effects that we wanted to achieve but could not with Tcl/Tk. We found that in tabular displays, like the memory display, using a grid of labels led to very bad performance in scrolling, resizing and moving the window, on Windows platforms. However, Jeffrey Hobbes' TkTable proved a good alternative, and performed quite well both on Windows and on our X based hosts. As I discuss below, the speed of loading the source window was an issue for a while, but we were mostly able to overcome that. There is still more work to be done, both refining the current UI, and adding useful features, but the hindrance here is simply time, and not the available tools.

Source Window

The central window in all of GDBTk is the source window (Figure 1). This contains the main menus for the application, the Toolbar for controlling program execution. The toolbar also contains buttons to access the most commonly used windows. The toolbar is actually a Windows style toolbar with buttons that raise when entered. It also has a series of Combo-boxes at the bottom that allow you to select a different file or function, and change the display mode from source to assembly or mixed source/assembly.



```
    39 #endif
    40 
    41 /*
    42  *----------------------------------------------------------------------
    43  *
    44  * main --
    45  *
    46  *      This is the main program for the application.
    47  *
    48  * Results:
    49  *      None: Tcl_Main never returns here, so this procedure never
    50  *      returns either.
    51  *
    52  * Side effects:
    53  *      Whatever the application does.
    54  *
    55  *----------------------------------------------------------------------
    56  */
    57 
    58 int
    59 main(argc, argv)
    60     int argc;           /* Number of command-line arguments. */
    61     char **argv;  /* Values of command-line arguments. */
-   62 {
    63 #ifdef TCL_XT_TEST
    64     XtToolkitInitialize();
    65 #endif
■   66     Tcl_Main(argc, argv, Tcl_AppInit);
-   67     return 0;        argv=(char **) 0xeffff834 prevent compiler warning. */
    68 }
    69 
    70 /*
    71  *----------------------------------------------------------------------
    72  *
argv=(char **) 0xeffff834
```

Figure 1: Source Window

Most of the work of the source window, however, is in the main text display area. The text in this display is divided by Tk Text widget tags into the break region, extending from the left edge to the end of the line numbers for lines that contain executable code, and the source region from the end of the line numbers to the line ends. The break region is sensitive to mouse clicks, which are used to set and remove breakpoints. The source region supports variable popup balloons, and context sensitive menus for dumping memory around the variable pointed at, and some other functions.

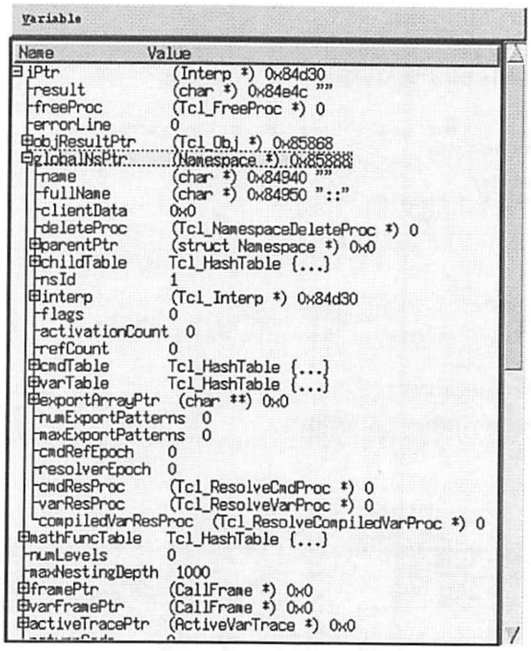
The source window required some effort to get it to have an acceptable speed. Source files can be quite large, and when you are stepping around among a number of files, any delay in loading the window can be quite annoying. We solved this in two ways. First, once a window is rendered, we cache the text widget. Then stepping from one file to another after the sources have been hit once is quite quick.

We also read in the files in C, and pass the data directly to the Text widget's C level command. Since we actually don't want Tcl to do substitutions on the data, this is actually a sensible thing to do. There is also a lookup that has to be done for each line - to determine whether it is executable or not. It is about 3x faster to do all this work in C than to use Tcl procs to read in the line, determine whether it is executable, and run the "widget insert" command.

At this point, the rendering of text is barely fast enough. It is not annoying, but it would make GDBTk feel more responsive if we could get another 2x in rendering the text. I have not yet experimented with using the private text widget routines to populate the widget directly, though that is the obvious next step. We may also be able to speed up gdb's symbol table lookup to determine whether the line is executable or not.

Variable Window

The variable window (Figure 2) actually took much more work to get right that it seems at first glance. The main problem was getting data out of GDB as quickly as necessary to update the window. You need not only to get all the variable values, but you need to get the block information, so you can don't duplicate shadowed variables, and of course check for changed variables so you can color them appropriately. Since all the variables had to be updated with each step, any delay was noticeable.



```
Name             Value
⊟iPtr               (Interp *) 0x84d30
 ├result            (char *) 0x84e4c ""
 ├freeProc          (Tcl_FreeProc *) 0
 ├errorLine         0
 ⊞objResultPtr      (Tcl_Obj *) 0x85868
 ⊟globalNsPtr       (Namespace *) 0x85888
  ├name             (char *) 0x84940 ""
  ├fullName         (char *) 0x84950 "::"
  ├clientData       0x0
  ├deleteProc       (Tcl_NamespaceDeleteProc *) 0
  ⊞parentPtr        (struct Namespace *) 0x0
  ⊞childTable       Tcl_HashTable {...}
  ├nsId             1
  ⊞interp           (Tcl_Interp *) 0x84d30
  ├flags            0
  ├activationCount   0
  ├refCount         0
  ⊞cmdTable         Tcl_HashTable {...}
  ⊞varTable         Tcl_HashTable {...}
  ⊞exportArrayPtr    (char **) 0x0
  ├numExportPatterns  0
  ├maxExportPatterns  0
  ├cmdRefEpoch      0
  ├resolverEpoch    0
  ├cmdResProc       (Tcl_ResolveCmdProc *) 0
  ├varResProc       (Tcl_ResolveVarProc *) 0
  └compiledVarResProc  (Tcl_ResolveCompiledVarProc *) 0
 ⊞mathFuncTable     Tcl_HashTable {...}
 ├numLevels         0
 ├maxNestingDepth    1000
 ⊞framePtr          (CallFrame *) 0x0
 ⊞varFramePtr       (CallFrame *) 0x0
 ⊞activeTracePtr    (ActiveVarTrace *) 0x0
```

Figure 2: Variable Window

The window also allows you to change the format of variables, and to open up structures. The type and value information is not as nicely laid out as it could be, mostly because the Tix TreeWidget's multi-column layout is not as flexible as we would like. However, it is quite fast at rendering its contents, which is important.

Target Selection Dialog

The target selection dialog is particularly important to embedded developers. It is where you configure everything that is needed to get the program loaded and running on your board. Because of this its contents vary widely among the various targets and transport mechanisms. This was one area in particular where Tcl/Tk really shone. The combination of the ease with which all the elements of the dialog could be configured from a tcl array, and the dynamic rescaling allowed by the Tk geometry managers, mean that we can use a Tcl array as a flat file database for all the target specific aspects of the dialog. The implementation is very easy to read, and understand, so that external developers can quite simply tune up the settings for their particular hardware.

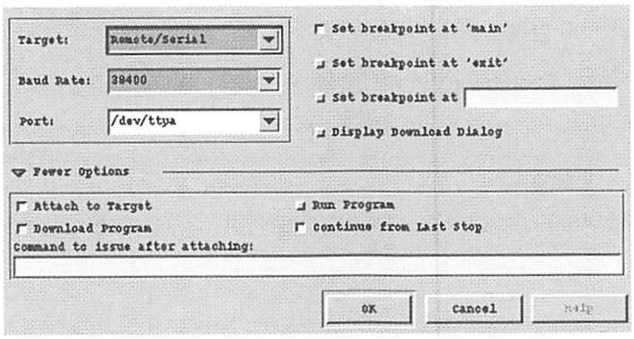

Target: Remote/Serial
Baud Rate: 38400
Port: /dev/ttya
Set breakpoint at 'main'
Set breakpoint at 'exit'
Set breakpoint at
Display Download Dialog
Fewer Options
Attach to Target
Download Program
Run Program
Continue from Last Stop
Command to issue after attaching:

Figure 3: Target Selection

Conclusions:

Incorporating Tcl/Tk into GDB posed substantial problems. In the end, the ease and flexibility with which you can accumulate results in Tcl, and the string handling functions it provides, made it possible to overcome the lack of a real C-callable API, without requiring a massive overhaul of GDB. However, we found putting an event driven application on top of a core which felt free to block at any time, was too fragile. It was necessary to change the way GDB worked in this area, to get a really stable application.

On the Tcl side, using Itcl was a very good choice for an application of this complexity. It allowed us to segregate our program into manageable pieces, and although we did not have a very involved design, the simple inheritance that we did use was very helpful. Between the IWidgets, some widgets from Tix, and other bits from the Net, we were able to realize most of the designs that we wanted to use in GDBTk. The only real lack is a complex tree/table widget.

Acknowledgments:

Stu Grossman deserves the initial credit (or blame) for getting GDBTk started. Martin Hunt deserves the lions share of the credit for the current state of the program. Keith Seitz made many valuable contributions, and was willing to chase obscure bugs into the bowels of GDB with awe-inspiring tenacity. Stan Shebs has consistently moved GDB itself towards the new millennium, and is responsible for sheparding into being the changes to core GDB needed to put GDBTk on a firm footing. And Andrew Cagney, Fernando Nasser, and Elena Zannoni have implemented most of these new features on the GDB.

Tcl/Tk: A Strong Basis for Complex Load Testing Systems

Ahmet Can Keskin, Till Immanuel Patzschke, Ernst von Voigt

Patzschke + Rasp Software AG

Abstract

This paper describes a Tcl/Tk-based load testing environment which was developed for Deutsche Telekom, Europe's largest telco carrier and online service provider. Deutsche Telekom uses the system to ensure a high quality of service and availability.

The paper explains the complex requirements of load testing and gives a detailed overview for the extensive use of Tcl/Tk within the system.

1 Introduction

The Center for Internet and Data Network Platforms (ZID), a division of Deutsche Telekom, provides the network infrastructure over which the largest Online Service in Germany, T-Online, is operated. Deutsche Telekom is Europe's largest and the world's third largest telecommunications carrier, and its online service has more that 3.3 million customers - which grew at a rate of 42 per cent during the last fiscal year.

ZID is responsible for ensuring that all hardware and software components of an access network function coherently and can withstand the heavy demands of hundreds of thousands of simultaneous users, both prior to deployment and during operation. ZID must also be able to deliver a guaranteed performance, bandwidth, and availability.

ZID has used the load testing technology for many years to ensure that the online services it provides maintain the Quality of Service (QoS) level that Deutsche Telekom's customers have come to expect. The objectives for those early test systems were functional testing, throughput and general performance testing for a BTX-based network. Recently, enhancements for the testing of the latest internet technologies, such as ISDN or ADSL, have been developed. Critical test factors were defect removal before deployment, and a guaranteed highly responsive system.

Since each new development stage has been accompanied by load testing, Deutsche Telekom was able to meet these goals and deliver a top quality service to its customers. The load testing technology used has evolved along with the network technology it is required to test.

Load testing is the general term used to describe placing a network infrastructure under stress to test its behavior in a production environment. We developed a fully scriptable load testing system based on Tcl/Tk. The basic principle in the design of this system is the Automatic User, abbreviated to AT - the short form is derived from the German term "Automatischer Teilnehmer" [1].

An AT simulates the actions of a human user (e.g. hitting web pages). Every AT can generate a certain load on the network, dependent on the connection (Modem, ISDN, ADSL). The load varies from 28.8 Kbit/s (Modem) over 64 Kbit/s (ISDN) to 768 Kbit/s (ADSL). The latest release of the AT system allows a maximum of 225 ATs to be run simultaneously, which are then managed and controlled with a graphical user interface, providing a single point of control and monitoring. The number of ATs used in a testing environment is virtually unlimited and only depends on the hardware configuration.

Both the control interface and the AT are implemented in Tcl/Tk. The scripting language Tcl proved to be an ideal platform for implementing the AT, which freed us from designing a new high level language for the specification of test scenarios and implementing a runtime environment to drive such scenarios. Tcl's mechanism of slave interpreters was sufficient to achieve these goals. Tcl/Tk also supported rapid development of the graphical control interface.

2 Background: Load Testing

To support its online services nationwide, Deutsche Telekom operates more than 180 POPs all over Germany. In each of the POPs, networks with specific hardware and software systems have been installed, which ensure the user's access to the internet. The introduction of new online services like ADSL requires the development of a new infrastructure for the access network, which has to be installed nationwide in all of the POPs. The requirements for such an access network are:

- High bandwidth
- Low costs
- High reliability

Designing such a network can be a challenging task. For providers of online services like Deutsche Telekom with 120 million internet connections per month, it is important to detect and eliminate deficiencies and performance bottlenecks before they deploy an access network nationwide. This is especially important because additional improvements can be very expensive, since all the POPs have to be upgraded and the users might not be able to use the services.

Deutsche Telekom has established two operation centers in Darmstadt and Ulm in order to test and optimize the entire system in a production-like environment prior to deployment. These centers are fully equipped with web servers, connection hardware, and all equipment for the planned network installation in a regional node (POP). Load testing is then performed using the AT on the complete network. Network components, including software, which allow access to the internet platform, are tested for their load-bearing capacity and stability.

Test results are used to make decisions on how best to optimize performance and deliver new functionality. Decisions such as replacing a router with a larger one are made. The test results are also used to verify whether or not the vendors components meet up to their claimed capabilities. The vendors providing equipment for the network often use the results produced by the AT (in conjunction with their own tools) to isolate defects, verify changes and fixes.

Once the developers at ZID are satisfied with the resulting network, it is deployed in more than 180 regional nodes throughout Germany. In addition the AT is also used in these locations to ensure their individual QoS once operational on a specific hardware basis. ZID relies heavily on the AT analysis tools to compare different load scenarios at different times of day. One of the major benefits of the AT is that all testing and analysis work is concentrated in one location and controlled from a single computer.

3 Design & Implementation

3.1 Requirements

In the past years, the ZID has been performing load tests with various tools for developing and optimizing their access networks. Experience has shown that load tests are most effective when the real conditions are simulated as closely as possible.

Let us consider a typical user, who wants to connect from his/her PC to the internet: He or she might be a subscriber to an Internet Service Provider performing transactions such as surfing the web, doing FTP file transfers, or sending e-mail via an SMTP client. Another typical user might be using an e-commerce site to make purchases or making queries to a web-enabled database application. Therefore, her or his essential activities are:

- Establishing a connection: The user dials his ISP's telephone number, connects, and logs into his account with a password.
- Doing online transactions.
- Closing the connection: The user logs off and closes the connection to his ISP.

To test the capacity of an access network (e.g. number of concurrent users), the load testing system had to perform realistic emulations for concurrent user activities, like HTTP, FTP, etc. Further requirements included:

- Single point of control (GUI)
- Measurement of performance criteria such as response time and throughput
- Flexible, yet easy-to-learn scripting language for defining test scenarios
- Scalability regarding the number of simulated users (AT)
- Extensibility, e.g. easy integration of additional network protocols (SNMP, RTSP, etc.)
- Online monitoring
- Distinct physical connection for each simulated user using different protocols (e.g. PPPoE)

3.2 Design Decisions

To meet the above requirements, we opted for an open platform providing support for almost all available hardware: Linux. In terms of a flexible scripting language, we looked at Perl, Python, and Tcl. Although Perl is pretty popular in the "system administration community" it isn't as easy-to-learn and handy as the other languages (especially for end-users.) A second important point was the GUI component - an area where Tcl/Tk is a natural choice, since both other languages lack native (i.e. out-of-the-box) GUI support and use for example Tk instead of a "native" solution.

Since the idea of the AT implies managing hundreds of different processes - including inter-process communication between them - easy-to-use and powerful network support was another important criterion. Last but not least the "glue" argument was very important. Tcl is the perfect tool for tieing different components together without creating monstrous extension libraries.

The AT's architecture is based on software agents that simulate a human user and are controlled centrally over a graphical user interface. (For details please refer to section 3.3: Architecture and System Components.)

Due to the flexibility of the Tcl/Tk environment, most of the entire system's components have been implemented using it:

- Rapid prototyping: When starting the development, we were unable to predict future technological advances and we did not know exactly which features the simulated user should support. It was thus important to have fully functional software agents whose features and functionality could be improved step by step.
- Description of scenarios: Tcl makes it easy to specify the activities of a test scenario because it is a high-level language providing all necessary control structures (if, while, etc.) to simulate every kind of sequence of a user's actions. This freed us from designing another scripting language for test scenarios and implementing a runtime environment.
- Visual Control: The management of the simulated user had to be possible from one central point with a graphical user interface - an easy job for Tk.
- Protocol modules: The simulated user should perform transactions based on the FTP and HTTP protocols. We were able to use existing Tcl packages that served as a basis to implement the desired functionality. Furthermore, Tcl's encapsulation of basic system functionality (like TCP/IP sockets) facilitated and simplified protocol implementation.
- Automatic User (AT): The simulation of a human user could be handled by a software agent which ran on a session host with a physical connection to the access network, and which was controllable from a central point (control host). Tcl's mechanisms for inter-process communication via sockets makes it easy to realize such distributed applications.

3.3 Architecture and System Components

Figure 1 shows an architectural overview of the load testing system. Each block in the diagram represents a logical software component.

A **load scenario** describes a complete set of transactions carried out by a specific number of users connected in a particular way to a network over a given period of time. Load testing involves managing a potentially large number of ATs executing load scenarios. It is especially important to have a means to control all aspects of testing from a central location.

The **AT controller** component allows the creation of an interactive environment for defining, driving, and coordinating a load test scenario. The AT controller is designed using a set of building blocks which allow a complete definition and management of a scenario. These blocks include:

- access to a repository of Session Scripts
- a means to define connections to the tested system
- configuration of the sessions
- session management during execution of the scenario

The **script repository** contains all the scripts that have been defined by the test designer. A given script can be assigned to one or more automated users for a given scenario. The AT provides example scripts which may be extended and/or parameterized to create unique automated users. Also the AT provides libraries implementing basic behavior patterns of real online users to reach a higher level of abstraction. A high level of parameterization is possible by allowing the script, to reflect the actual behavior of a typical user without having to create a new script even when the basic actions taken are not the same. For instance, a time-out can be specified to simulate a user cancelling a web page fetch after 5 or 10 seconds. Pauses (or sleeps) can also be defined. The specific URLs fetched or the username/password required as input for certain operations can all be parameterized. The parameterization is done by the test designer either programmatically to the AT controller or via a GUI.

An **automated user** simulates a real user performing typical operations. A typical user might be a subscriber to an Internet Service Provider carrying out transactions such as surfing the web, doing HTTP GET or POST operations, downloading files using FTP, or sending e-mail via an SMTP client. Another typical user might be using an e-commerce site to make purchases or make queries to a database application through a web front-

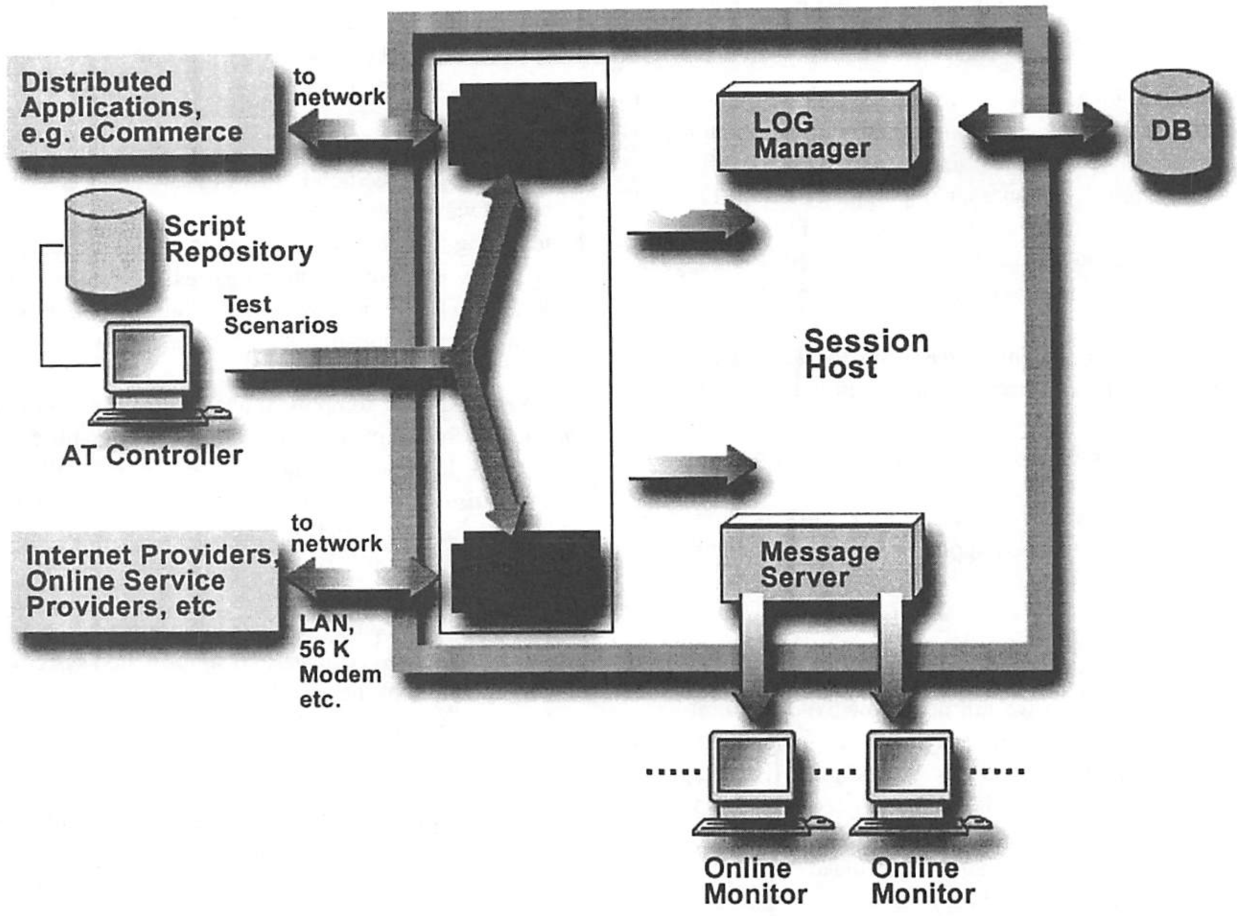


Figure 1: Architecture of the Automatic User

end. The AT creates a run-time environment in which a session script is run performing a work session. The AT builds the connection to the network, carries out the operations specified in the session script and disconnects from the network at the end of the session. Since the AT can be configured to connect via an analog or ISDN modem, a LAN or WAN, amongst others, the throughput measured is what a real user could expect to see. The AT is also responsible for forwarding test results to the message server and the log manager for monitoring and analysis. It also provides status information to the Session Control upon request, including such things as whether a session is currently executing, whether a script is loaded, etc.

The **log manager** receives messages from the various ATs running on a session host [2]. It adds a time-stamp to each entry and, depending on any filtering of messages that has been set, writes the message to the designated logging facility. Messages may be filtered based on the message type. Current types of messages are: trace, log, debug, performance, error.

Like the log manager, the message server receives messages from the various ATs running on a session host. It filters test results and passes them to the connected online monitors where the tester can review the progress of the load scenario and take actions based on the results. Filtering is specified by the online monitors. For instance, an online monitor may specify that it is only to receive trace messages only.

The **online monitor** makes it possible to take action on tests as they take place. For instance, it may be necessary to create additional load over a specific connection to stress the system in a new way based on the results being produced by the currently running sessions. Multiple online monitors may be connected to a session host which allow customized viewing of results. The number of monitors and the purpose of each is specified by the test designer. The monitors connect to the message server running on the session host. Each online monitor can filter the different types of messages processing only those of interest to it. A monitor specifies to the message server what types of messages it wishes to receive.

Several online monitors can be created, each tailored to the specific needs of a test installation. One example is a specialized online monitor listening to trace messages only, sending an e-mail and/or SMS message when a URL specified is not reachable.

Another likely configuration would use an online monitor that views only debug messages which need to be handled by the creator of the load scenario and another online monitor that views only performance messages used by the network architect to identify bottlenecks and refine the network configuration.

All online monitors can be connected to a graphic interface providing dynamic graphic views on result data. The entire dynamic interface is built using the Tcl/Tk based product GIPSY [4], allowing easy customizability and creation of dynamic data visualization.

3.4 Distributed Components and Communication

Since Tcl has easy-to-use communication commands, building a distributed client/server system is simple. On top of standard Tcl we implemented a communications layer as basis for RPC and simple data streaming (for fast transmission of status data.) Additionally an "Application Name Server" (ANS) provides host/port lookup services.

Tcl-DP [3] appeared too "heavy-weight" for the above purpose, so we decided to use "vanilla" Tcl to realize the entire communication layer.

Needless to say that all components of the AT use the communications layer, allowing easy expansion and distribution of functionality. System Configuration

3.5 System Configuration

A typical installation of the load testing system consists of one control host and a number of session hosts. The control and session hosts are connected through an ethernet LAN, which we call AT backbone. Several ATs run on each session host. The number of ATs per host depends on the connection to the ISP or the Internetwork (ISDN, Ethernet, ADSL, Modem etc.).

The current configuration at ZID uses 4 4-port ethernet network adapters per PC emulating ADSL lines to an Access Concentrator (AC), providing 15 session and one backbone port. The backbone connects 15 PC running ATs to one AT controller, acting as single -point-of-control for the entire system.

4 Automatic User

As stated above, the AT simulates a real user performing online actions like HTTP get or FTP file transfers. Implemented as a software agent, the AT is built upon four major modules: communication, connection, logging and script engine.

The communication module encapsulates all communication aspects like a RPC server socket, connecting to the controller or the message server and sending them status and log messages. The AT receives and executes commands from the AT control via the RPC server socket.

The connection module provides all functions to connect to a network and authenticates a user with his name and password. The logging module implements functions to log all actions of an AT to a plain file, a database, send log messages to syslog daemon or the log server. Finally the script engine provides a runtime environment to drive session scripts.

4.1 Session Script

In order to simplify the development of the session scripts and to reach a higher level of abstraction, Tcl libraries are provided. They already implement a basic behavior. This is illustrated in the example below. Listing 1 shows a sample session script. It is a simple example simulating a user who establishes a connection to the ISP via PPPoE, gets some URLs and disconnects.

```
# Sample Session Script
#
set ::url1 "http://192.9.220.240/test/600k.gif"
set ::url2 "ftp://193.156.56.10/2400k"

# set basic connection parameters
access::config -trymax 3 -trydelay 300

# define bandwidth paramters, i.e. if
# bandwidth<30 and times>=4 ...
access::perf::config -threslow 30 -maxnumlow 4

# session main
proc run {} {
    # connect using PPPoE
    access::connect -p pppoe -user joe \
    -passwd pizza
    # get 600k.gif via HTTP
    access::perf::get $::url1
    # get 2400k via FTP
    access::perf::get $::url2
    # end session - disconnect
    access::disconnect
}
```

Listing 1: A sample session script

If an action (e.g. `access::connect`) fails, it is repeated several times (`::try(max)`.) If it still fails, an alert is sent to the online monitors and the log manager. Then the action is repeated after a defined number of seconds (here `::try(delay)`) seconds) until it is successful.

Additionally, a threshold value for the data transfer rate can be defined (`::thres(perf)`.) If the transfer rate drops below the threshold value more often then a defined limit (`::thres(num)`), an alert is raised.

4.2 Script Engine

The script engine is a Tcl module providing a runtime environment for session scripts. To drive a session script the script engine first reads the Tcl source of the session script from the script repository. Then a slave interpreter is created and initialized. In the initialization phase local variables are declared in the slave interpreter, aliases to commands of the master interpreter are created, protocol extensions loaded. Finally the session script is evaluated within the slave interpreter.

To start a session, the script engine invokes the `run` procedure of the session script. Stopping a session means deleting the slave interpreter ignoring the resulting error messages.

Session scripts are written in Tcl using behaviour libraries. Each script has a `run` procedure. The `run` procedure may be executed multiple times to simulate multiple consecutive sessions. All statements outside of the run procedure are executed once per session.

The script writer may define trace messages that will be sent to the message server and log server. Within the script any combination of transactions using the supported protocols can be performed as well as all standard Tcl operations.

It is also possible to set variables in the script prior to running it. This allows customization of a single script logic for different users who perform the same operations but on different targets including: number of run repetitions, transaction time-out intervals, and level of tracing.

For FTP and HTTP we use extension implemented in Tcl. The http extension is built on top of the standard HTTP package coming with Tcl. Ftp is built on the FTP package of Steffan Traeger [2]. The FTP and HTTP packages had to be modified in order to measure the response time and throughput and to provide distinct time-out mechanisms for the connect and for the data transfer phase.

5 AT Controller

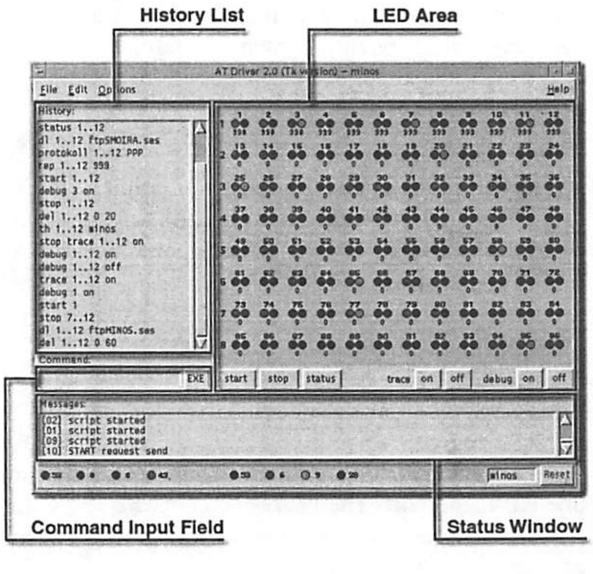
The management and control of both the ATs and sessions are accomplished via a graphical user interface (figure 2) which gives the user an overview over the ATs states. The user may start any test scenario and monitor the load tests and their results.

The controller consists of an LED area that represents the ATs, a command input field, a history list, and a status window.

The LED area displays three states for every AT: script status, connect state and a "runs left" counter. The script LED denotes whether a script has been loaded, started or stopped. The connect state indicates one of the following: if the AT is connecting to a network, if it is already connected, if it is about to disconnect or if it is already disconnected.

Controlling the AT means choosing from two options: Direct input using the command input field or selecting the ATs with the mouse cursor and clicking the command button.

The LED area is sensitive and the user may choose the AT to which a command is sent using the mouse. Additionally, he may re-activate commands from the history list. The execution of a command is shown in the status window. The controller is connected to the ATs and may send any command to them via RPC.



```
status 1..12
dl 1..12 ftpSMOIRA.ses
protokoll 1..12 PPP
rep 1..12 999
start 1..12
debug 3 on
stop 1..12
del 1..12 0 20
th 1..12 minos
stop trace 1..12 on
debug 1..12 on
debug 1..12 off
trace 1..12 on
debug 1 on
start 1
stop 7..12
dl 1..12 ftpMINOS.ses
del 1..12 0 60
```

```
[02] script started
[01] script started
[09] script started
[10] START request send
```

Figure 2: AT's user interface. For description, please see text above.

6 Experiences with Tcl/Tk

This paper described a fully scriptable load testing application for networks. All crucial components are implemented with Tcl/Tk. The use of Tcl/Tk allowed a development process that is best characterized with "development by prototyping", and thus enabled us to present a working version to the customer very early in the project. As the project progressed, we could build fully working prototypes and test them under "real world conditions". This helped us to better understand the requirements and - even more important - to find and eliminate errors in the design very early. Last, but not least, we could implement changes which our customer requested after the design phase with an effort bearable for both sides.

Another important aspect of the "development by prototyping" was that the testing engineers were able to improve the design of the access network based on the test results of prototypes, even in an early development phase. A good example is the ADSL modem-access implementation history. Any access method involving substantial user configuration of their modem or personal computer can hinder rapid, widespread consumer adoption of high-speed services such as ADSL. The first approach, which was provided by a well-known vendor of network components, was a combination of a DHCP client and a HTTP-Browser. The user was required to configure DHCP on his Windows client and open a URL with his Browser to get a login page. After typing user name and password the connection to the internet was established.

As the test engineers began load tests, they detected instabilities of the access network components. The implementation was neither stable nor would it allow more than 28 concurrent logons. These results were reproducible so ZID could reject this approach and force its network solution provider to develop an alternative solution.

The next and up to now final solution for client access method was point to point over ethernet (PPPoE). Because PPPoE had been a rather new protocol there were not clients available at that time, especially not for the Linux operating system. So, we had to implement a client on our own and integrate it into the load testing system. This kind of change was only possible due to Tcl's excellent "gluing" capabilities (and the flexible architecture, of course), allowing ourselves to keep most of the existing system without any changes for the new protocol.

During later prototyping phases the system was able to detect problems concerning the stability and the bandwidth of the whole access network solution. ZID's hardware vendor and solution provider had to admit that neither the ATM switching component nor the PPPoE server performed within specification (50Mbit/s vs. 150Mbit/s, 56 vs. 90 sessions.)

Another important aspect for our customer was that we did not modify the Tcl kernel. All of the requirements have been solved with standard Tcl/Tk. C was only used in performance critical areas like the PPPoE protocol handling (i.e. handling Ethernet packets.) Using standards - like Tcl/Tk - without modifications proved to be an important issue, especially regarding the system's acceptance.

It proved that Tcl/Tk was and is the ideal solution for the implementation of the AT, since we were able to meet the customer's requirements effectively and react in a timely manner to design changes during the project. Tcl offered an easy-to-learn and flexible language that allowed us to provide a full featured language gaining more and more enthusiasm at ZID (all of the test engineers are non-programmers.) All that was possible without developing a language ourselves by simply using what was already there.

Tcl/Tk helped us to develop a commercial product in a short time and - even more important - to release it to our customer right in time. From our customer's point of view, Tcl/Tk helps in achieving his mission critical goals with minimal effort.

References

[1] AT and sm@rtTest

The Automatic User described in this paper is marketed by interNetwork AG.

More information may be obtained from their web site under http://www.internetwork-ag.de

[2] FTP Library Package

The FTP Library Package ftp_lib provides the client side of the File Transfer Protocol (FTP). The package extends Tcl/Tk with commands to support the file transfer protocol like OPEN, CLOSE, LIST, PUT, GET, REGET etc. It's used either to add FTP ability to existing Tcl/Tk applications or to create small FTP scripts that perform tasks without user interaction. It allows automatic up/download processes even up to the mirroring of complete FTP sites.

Information on the web:
http://home.t-online.de/home/Steffen.Traeger/tindexe.htm

[3] Tcl-DP

Mike Perham, Brian C. Smith, Tibor Jánosi. "*Redesigning Tcl-TP*" in Proceedings of the fifth Tcl/Tk Workshop. July 1997.

[4] GIPSY

GIPSY is a Tcl/tk-based platform-independent visualization software which is also provided from PRS.

Information on the web:
http://www.prs.de/int/products/gipsy

Address of the Authors:

Patzschke + Rasp Software AG
Bierstadter Straße 7
D-65189 Wiesbaden

Germany

www.prs.de - info@prs.de

Using Tcl To Build A Buzzword* Compliant Environment That Glues Together Legacy Analysis Programs

Carsten H. Lawrenz, Rajkumar C. Madhuram,
Siemens Westinghouse Power Corporation, Orlando, Florida
`{Carsten.Lawrenz,Rajkumar.Madhuram}@swpc.siemens.com`

Abstract. The Siemens Integrated Design (SID) Environment is a system that allows engineers to link together many legacy computer programs. This capability provides significant reduction in effort for defining the conceptual design of electrical generators. The SID environment is a generic tool for running all types of analysis programs (methods) as well as managing their associated data. Methods are plugged into the environment in a simplified fashion by using a well-defined interface. Any features that are added to the environment immediately benefit all methods. Data can be shared between remote sites through an in-house developed, java based, replication server. This paper discusses how Tcl was used to develop the SID Environment and why it was the best choice for our application.

* *Buzzwords: Scalable, Multi User, Client Server, Distributed, Cross Platform, Customizable.*

1. Introduction

1.1 Business Case

Siemens Westinghouse relies on various complex computer calculations for designing electrical generators. Various computer programs (methods) written in FORTRAN and other languages have been used over time. Because each method addressed specific aspects of generator design, they existed as standalone entities. Running each method required the manual creation of input files and the manual extraction of output data to prepare it as the input for other methods. More complications arose when trying to incorporate the more contemporary functionality of spreadsheets. This process, and its many iterations, when performed by several different design teams, resulted in several manual hand-offs of information. Furthermore, this created an immense paper trail, data integrity issues, and time and effort spent trying to keep the design teams synchronized.

There was an effort to reduce the time required to conceptually design an electrical generator from 180 Days and 10 engineers to 18 Days and 3 engineers. The project was incrementally funded; therefore quick turn around time was required.

Although most of our user platform was UNIX based, we knew we may want to use the Windows platform some time in the future. The computer languages with GUI support that ran at the time (1995) on both platforms were limited. We evaluated a third party GUI library (Galaxy) which was C based. We found its use of coding too difficult to develop a prototype quickly. Java was still only known as Coffee. By chance, the authors discovered Tcl/Tk. Although Tcl was not officially available on Windows at the time, various Windows versions did exist.

Siemens Westinghouse designed an architecture that enabled methods to be easily plugged in, instantly sharing data among other methods. Using Tcl lent itself to fast prototyping and development. Six months into development, engineers were able to start using the newly created environment.

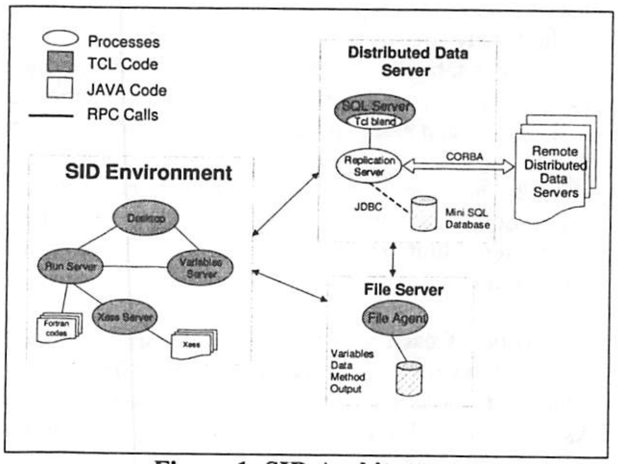


Figure 1: SID Architecture

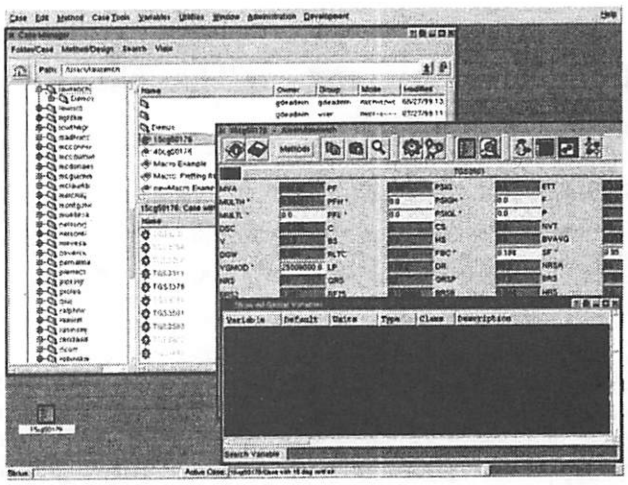
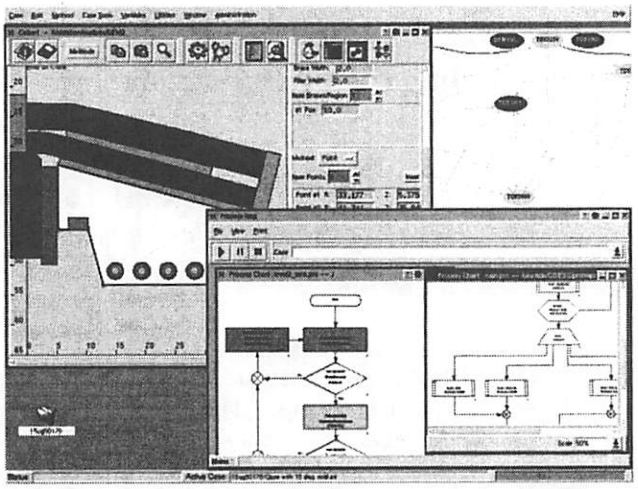



Figure 2: Snapshots of the SID Environment
(Left: Case Manager, Input Screen, and Variable table. Right: Dynamic Input Screen, Data Flow and Process Maps)

1.2 Basic Usage

The SID Environment simplifies the conceptual design process by automating and linking manual tasks. Users attend a one day training class during which they're ID's are registered in the SID database. Once registered, typing 'sid' in a command window launches the environment.

Each user works in their own unique data set that we call a case. These cases are collected in folders and are presented in a hierarchical format similar to a file manager. The folders and cases also mimic UNIX type file access permissions, which the user can modify. Any a number of methods are associated with each case. The user opens a case by selecting a method. The case is then locked to disable access by other users.

The case is displayed to the user as a window. Within this window the input screens for each associated method may be displayed. A method screen, by default, is a table of variable names. The environment allows customized input screens for each method as well. Users can select on any variable's entry box and change its value. Validations of input data for each variable are provided. A popup menu provides descriptions for each variable. The variables' descriptions, units and type are all defined in a global variable file.

The scope of each variable is global within the case, which allows methods to communicate. The user may select to run one method or several methods in series. As each method runs, the environment automatically updates the variables within the scope of the case. This transfer of data is accomplished by the use of input and output forms which are detailed below.

The environment also provides many utilities for viewing the various output types (PDF, postscript, plots, HTML) created by the methods. All output viewing is launched from a data viewer tool. This allows the less computer savvy users to be very productive.

2. Architecture

The architecture of SID Environment is outlined in Figure 1. The SID environment is a collection of servers and the desktop client. The desktop client (shown in Figure 2) handles most of the user interactions with the system. It communicates with several other servers for performing various functions.

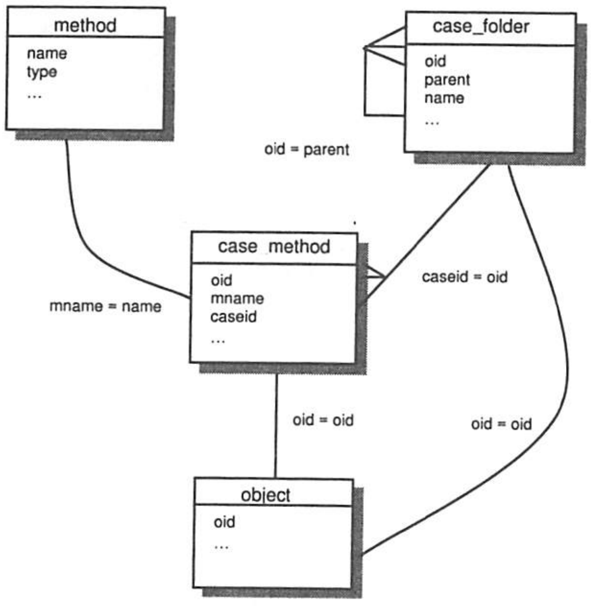


Figure 3: E-R diagram for case and method tables

Variables are the most widely used logical entities in the system. They are stored in a data file in key-value ordered pairs. Each method takes input from a set of variables and generates output that is returned. For every design calculation or analysis, engineers use a set of methods, which we group as a case. In essence, a case is the set of methods and their related variables.

Cases have to be managed in a reliable and fail-safe manner, especially in a multi-user environment. Hence, we employed a mini sql (mSQL) database [Hugh99] to hold the information about the cases, methods, users and the relationships between them. Cases are organized in the fashion of a UNIX file system, with a hierarchy of folders and permissions to control access. Each case and each method in a case are uniquely identified by an object id (oid) and its parents oid. The relations between these entities are shown in Figure 3.
As the need for exchanging data between engineers in remote sites surfaced, we wanted some of the folders to be shared between all the sites. In order to implement such a distributed system, we decided to use CORBA because it has established itself as a reliable and robust way to build distributed applications [Wolf98]. A replication server was coded in Java, which uses JDBC to communicate with the database. Tcl blend was used with the data server to access the replication server objects.

It was also required to port the system to Windows NT. In order to give uniform access to the case data, a file agent was required. It handles data movement to and from a case on behalf of the user into the central data repository. This mechanism also provides an added level of security, since all the data is owned by the file agent and access is only allowed through this file agent.

3. Software Development

3.1 Debugging

The fact that Tcl is as fully interpreted language lends itself well to software development. Many programmers fault Tcl because it does not provide syntax checking like other languages such as C. However, in the Tcl mode of programming the developer is able (with tools like TkInspect) to dynamically modify the code while the program is running. Furthermore, modified code can easily be reloaded into the interpreter by re-sourcing the code without having to exit the program. Also, bug fixes can be copied out to the production area without having to take the whole environment down. We found this approach to programming to be much quicker than the code, compile and debug method.

Tcl's error handling is more graceful than C or Java also. Most errors aren't fatal, i.e. the environment usually does not crash when errors are encountered. We overrode the error handler with a dialog box, that allows the user to email the developer a description of what caused the bug and a stack trace. This usually provides the developer enough information to determine the cause of the error.

3.2 Coding

SID requires relatively few lines of Tcl code. This is advantageous because the development team is only allotted 1½ man years per year for environment maintenance. The SID environment contains over 115 thousand lines of code and is maintained by only two developers. This smaller body of code also lends itself to utilities such as Concurrent Versioning System (CVS). Much of the coding is almost self-documenting; understanding code logic comes quickly. It's conceivable that to achieve this same functionality using C or Java could require about ten times as many lines of code.

The Tcl auto loading of methods also helps developers organize their source files in a comprehensive manner either by components or functionality. In the SID environment, our source directories are several levels deep.

3.3 Ease of Learning

The development team relied on contract labor to help meet the demands of the project in the early stages. The developers that were hired to program the environment did not have any prior knowledge of Tcl/Tk. However, it was easy to get motivated programmers up to speed quickly and start development. One observation is that experienced C programmers do not necessarily make good Tcl programmers. Strings and lists manipulations are so powerful and easily handled in Tcl. Tcl programming requires users to accept a paradigm shift while solving problems.

3.4 Reusability

Adding a method to the environment requires insertion of a record in the method table (Figure 3) and the creation of a file *(method.screen)* that lists all of the input and output variables used by that method. SID parses the list, and by default, a generic input screen is created for that method (See input screen in Figure 2). Several other applications source this same file. For example, there is a web server script that reads this file to create an input/output dictionary the methods online documentation. There is also an administrative application that reads every methods screen file and creates a matrix of all variables and how they are used by each method. This matrix allows SID to determine if other method's data has been invalidated due to the change of a variable's value. This simplifies method administration because all information is kept consistent and concurrent.

4. Most used Features of Tcl

4.1 Graphical User interface

The SID environment relies heavily on GUI components. The environment creates a large number of entry widgets, which are used for data entry. There are also a lot of bindings attached to these widgets to handle data validation. The GUI commands blend well with the source code because of their simplicity compared to other languages such as Java [WeFr97]. Very complicated and large input screens are easily handled by the Tk extension, as also witnessed in other large applications [Angel98] [DeCl97]. Our own experience has shown that dynamically creating an input screen with over 100 entries in Java required minutes compared to seconds in Tcl.

Some methods are preprocessors to complex Finite Element Analysis (FEA) models. These methods have customized input screens with simplified graphics of the various model components. These graphic elements may be adjusted manually to modify variables or they can redraw themselves to reflect the value of variables (See graphic input screen in Figure 2).

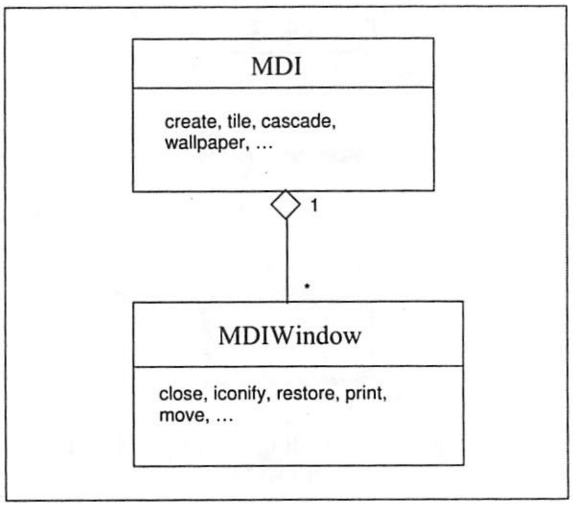

MDI
create, tile, cascade, wallpaper, ...
1
*
MDIWindow
close, iconify, restore, print, move, ...

Figure 4: Tix MDI classes used in SID

We also created our own Multiple Document Interface (MDI) widgets on top of the Tix framework. The MDI system consists of the MDI widget and the MDIWindow widget (Figure 4). The MDI widget is a container widget that contains MDIWindow widgets. All the tools within SID environment are built using the MDIWindow widget. The advantages of such a scheme are many: 1. It enforces uniformity in all the windows and provides access to features provided by the MDI system and 2. It provides a compact environment where all SID related components are held together.

Users can create new customized input screens. By placing local versions of the screen files in a pre-defined directory. These files override the production version of the files. This allows developers to test functionality without having to check out a entire local copy of the environment. Once the new screen has been tested it can be copied out to the production area to be shared by all users.

The user can customize the look and feel of the environment. There are many configurable options like wallpaper, fonts, background color etc. In addition to

GUI, many options like viewers, sound etc., can be customized.

4.2 Strings and List Processing

We use the string processing capabilities of Tcl in SID. It is easy to create meta-languages for various purposes. For example, we developed a "forms" language, which is Tcl with a few abstractions that provide a generic interface to the legacy codes. An input form (iform) is a template for creating input (Figure 5) and an output form (oform) is used to get the output values back into SID. It is an easy and yet very powerful method of parsing the input and output.

Another example where the string processing capabilities were heavily used was in creation of a

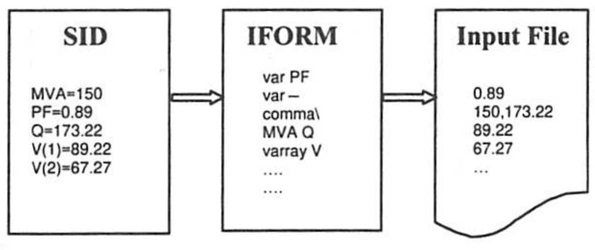


Figure 5: Mapping of SID data to Input File

macro language. A macro is a sequence of SID actions that an engineer can use for design work. We created an environment where macros can be edited and run, complete with debugging options like stepping, watch and breakpoints. The language of choice for the macro was obviously Tcl and we supplemented it with some commands like RUN, GRAPH, CALL etc., to provide some higher level abstraction. For this, we used regular substitutions (`regsub` command). We avoided using slave interpreters since a tight integration with the data in the environment was necessary.

In order to perform dynamic highlighting when a macro is run and also for debugging, a parser is needed. Conventionally, one would use lex and yacc to specify the grammar and then compile it with C to get a parser. However, we decided to use the powerful `regsub` command in an iterative fashion. Whenever a macro is run, it is first pre-processed in a two-phase method. We first substitute markers of the form *@@Mark[nn]@@* instead of newline characters, where *nn* stands for the current line number. We also handle line continuations by introducing special markers. In the next phase, all the markers of the above form are substituted with a *macroPhase2* command and a check to see if it was halted. The command *macroPhase2* is called with the current line number and a pointer to the macro environment. It handles things like highlighting the current line in the editor, handling break points and also acts as a state machine to put the macro engine to the next state based on the user action (stop, reset, run etc.). Since a macro is typically a small script (less than 100 lines), the pre-processing is fast (~0.5 secs/100 lines of code) and hardly noticeable. Tcl enabled us to create a fairly robust macro system within a matter of days, which would not be possible had we chosen a different language.

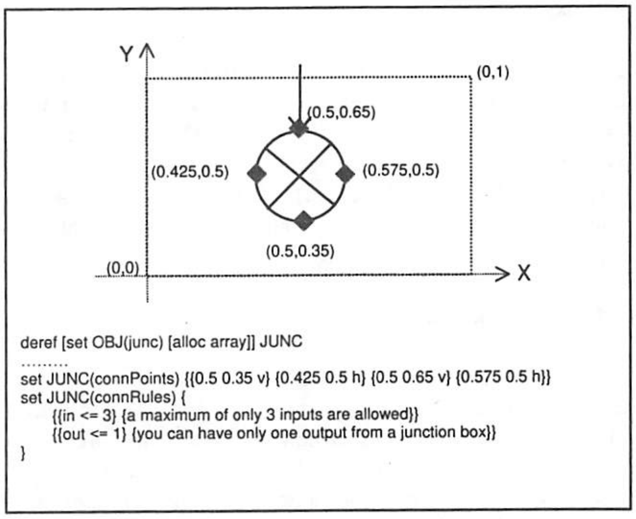
Large lists are handled efficiently in Tcl from version 8.0 onwards. Consequently, we found a tremendous increase in performance of SID. The number of variables in a typical case can go above 5000. We store these variables in global arrays using `array set` command, which is many times faster than iterating through the list and storing them individually.

4.3 Global Variables

Many of the routines in SID rely on pointers, which are actually references to global arrays. Pointers are useful in creating complex data structures. They also provide an extremely convenient and clean way of keeping track of state information within an environment as large as SID. We use a single global variable GV that provides pointers to all the information about the current state of the environment. This makes it easy to organize the data in a hierarchical structure. For example, in order to determine if a module named tgs8000 has valid data in the active case, we could traverse like this

```
deref $GV(active_case) case;
deref $case(module_ptr) module_list
deref $module_list(TGS8000) module

if $module(valid) {
  ......
}
```

The pointer mechanism comprises commands `struct`, `alloc`, `free` and `deref`. The `struct` command can be used to create data types similar to that in C. Whenever the `alloc` routine is called, it creates a global variable of the form _mem<*nn*> where *nn* stands for a sequence number generated using a counter. A pointer is returned, which is of the form <level>#_mem<*nn*>. The `deref` command creates an alias for the global variable referenced by the pointer. In the absence of object oriented constructs (i.e, without Itcl), the pointer mechanism provided a way of neatly packaging different components inside SID. Also, we wrote a simple script that would go through all the `struct` definitions and create html documents. It serves as a good reference document to the internal data structures of the system.



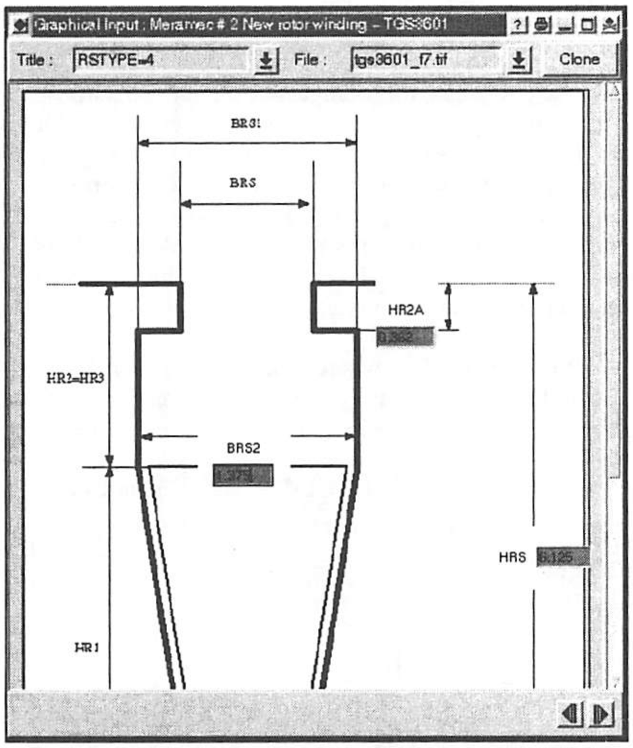
```
deref [set OBJ(junc) [alloc array]] JUNC
........
set JUNC(connPoints) {{0.5 0.35 v} {0.425 0.5 h} {0.5 0.65 v} {0.575 0.5 h}}
set JUNC(connRules) {
      {{in <= 3} {a maximum of only 3 inputs are allowed}}
      {{out <= 1} {you can have only one output from a junction box}}
}
```

Figure 6: Specification of a process chart primitive

Variable tracing is another aspect of Tcl that we used in SID. We use it in macros to associate the pseudo variable names with the real variables so that the internal mechanisms are hidden from the user. It is used in several places when we need to fill certain lists so that it remains consistent with the context of the application. One interesting lesson that we learned was to avoid variable tracing if it is needed only in certain instances when the variable in question is accessed. Trying to have a global variable that flags the tracing on and off creates problems that are hard to debug. One instance is when an error causes the interpreter to spiral out of a routine before the flag is not restored to its proper state.

4.4 Graphics Capability

The canvas widget is the only one that provides drawing capabilities. Nevertheless, it is very powerful


Graphical Input : Meramec # 2 New rotor winding - TGS3601

Title : RSTYPE=4 File : tgs3601_f7.tif

Figure 7: Snapshot of Graphical Input Screen

and we used it in components like process maps, data flow diagrams and custom methods. Process maps are basically flowcharts that represent an engineering process. The goal was to create and represent these processes inside the SID environment. First we investigated commercial charting programs, but these did not suit our requirements. Also, the Slate package [RL98] was not available at that time. Therefore, we decided to use the canvas widget to create one. We developed a set of primitives such as decision box, process box, sub-process box, etc. The whole process chart is represented as a flow chart structure. The sub-process primitives have pointers to the graphs of the corresponding sub-processes. Thus, the result was a hierarchy of graphs for a given process. All of these were neatly handled by the pointer mechanism that was discussed earlier.

Another interesting aspect of the process charts is the links that connect the different primitives. We introduced the notion of connection points, which are pre-defined positions around the primitive from which a link could be drawn. When a user drags the mouse to create a link, we search for the nearest connection point that is available. Also, every primitive has constraints on the connection points. For example, the start/end box could have only one output or one input and the decision box can have only one input and two outputs. Again, the scripting nature of Tcl "came to the rescue" in describing and checking the constraints. For example, the junction box object has a description similar to the one shown in Figure 6.

The co-ordinates of the connection points are relative to a hypothetical 1.0x1.0 bounding box of the primitive. The v and h indicates that only a vertical or horizontal connection is allowed at that point respectively. While parsing the rules, "in" is substituted with the number of total inputs (+1 if the current connection point is being considered for an input) and "out" is substituted correspondingly. The rule engine goes through each rule and evaluates the rule expression. If it fails, it displays the corresponding error message and returns. Tcl makes the process very generic and simple. One could even have rules such as "in+out <= 1" (in case of start/end box).

Each of the primitives and links has a pointer to an array that contains information about the object. The tags for each are also named accordingly. The tag for a primitive may look like obj_0#_mem72 and that of a connection, like con_0#mem66. Using `regexp` and `deref` commands, we get a quick access to the properties of the object when it is selected for various reasons like cut/paste, delete etc. The editor also provides powerful features like cut/copy/paste and undo/redo operations.

Once we got the process map editor in place, incorporating it into the environment was simple. Next, we were also able to make the process "run". A flashing circle beside the primitive indicates that it is currently executing. The handling of multiple charts in each case was done using our MDI window widget.

We use the BLT graph widgets for plotting graphs within the environment. The package PlPlot [PL99] is used to generate high quality engineering graphs without having the need to display on the screen.

We also provide graphical input screens, which are entry widgets overlaid on a gif image of a drawing (Figure 7). This provides an intuitive interface, especially for the novice users to enter values for various input variables in the method.

4.5 Networking/Communications

Communication between the various networked components is done using the Tcl-Dp package. The RPC mechanism is robust and provides a simple interface for servers and clients.

The servers within the SID environment are spawned on every invocation. It is not possible to assign fixed port numbers to each server since more than one user can be using SID (with remote displays) on a same machine. In order to solve this problem, whenever a server is spawned, it searches for a free port and stores the port number information in a registry file. This file serves as a directory for the various network components.

Interactions between the servers can be quite complex. Figure 8 shows the events that take place when the user clicks on the RUN button. Communications between servers take place through RPC and the status of other processes are monitored by the UNIX signal trapping commands provided by extended Tcl.

The data server provides replication services. Because, replicating all the cases involves huge network traffic and is largely unnecessary, we identified a subset of folders that would be replicated. There are two

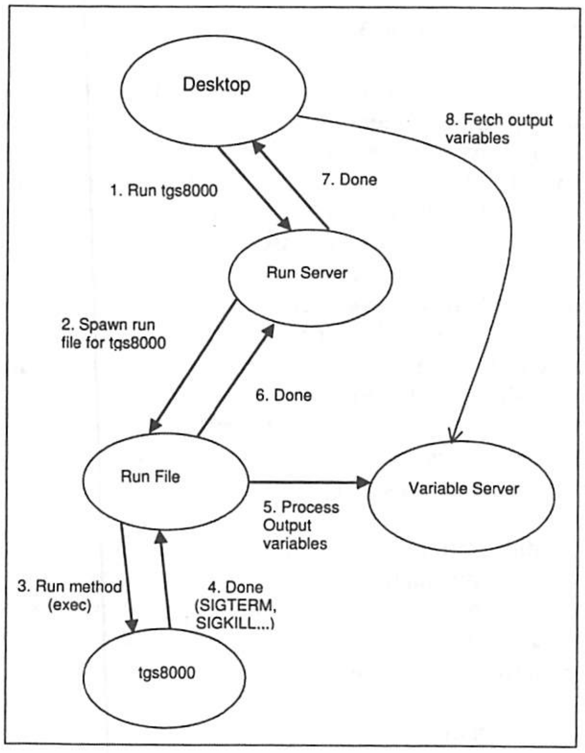

Desktop
8. Fetch output variables
1. Run tgs8000
7. Done
Run Server
2. Spawn run file for tgs8000
6. Done
Run File
Variable Server
5. Process Output variables
3. Run method (exec)
4. Done (SIGTERM, SIGKILL...)
tgs8000

Figure 8: Sequence of events during a method run

replication modes, synchronous and asynchronous. In synchronous replication, the action has to be completed in all sites before proceeding. For example, when someone tries to open a case, it needs to be locked. The sql command to lock the record is passed on to all the sites. In each site, the command is supplied to the msql daemon, via JDBC. Only when it is successful in all sites, the locking is valid. On the other hand, asynchronous replication passes on the command and simply returns. Each site holds a command queue (java layer) which keeps trying until the operation succeeds. Replication of case data is handled using FTP. We tested the replication between three sites (Orlando and Charlotte in the U.S and Muelheim in Germany). We found that the network bandwidth, which we hope will improve, was the only bottleneck. It works exceptionally well and as planned, a testimony to Tcl's claim as a glue language. We use xess spreadsheets [Ais99] for some of our methods. There is a Tcl interface for the xess spreadsheets that we use to communicate between the spreadsheets and the SID environment in the unix platform. We communicate with Microsoft Excel spreadsheets on NT using the package tcom that exposes COM interfaces. We use a Tcl implementation of SMTP that we found in the newsgroup for all our mail purposes within the system. It works across platforms and is reliable.

4.6 Cross Platform Deployment

We recently started porting the SID environment to NT after all of the required extensions were available. Surprisingly, there were few coding changes required to bring up the environment. All GUI components worked extremely well on the first attempt. Early in the development, we made a conscious effort to remove as many `exec` commands with in the Tcl code. The majority of changes were related to file permissions and sharing between UNIX and NT. This problem was tackled by using SAMBA and by writing a small File Agent that handles most file duties on behalf of the user. This gave us an added benefit, the File Agent becomes the owner of all files in the repository thereby restricting access by general users.

We did notice some reduced performance when creating large input screens running on a Pentium II Windows machine. Also, some of our input forms would not display properly unless we added a few well placed `update` commands.

5. Conclusion

Our experience has been that we're able to add any desired feature into SID with Tcl. Most of our design was incremental and Tcl provided the flexibility to meet our goals. We could create prototypes quickly and incorporate them into the environment. Relying on many extensions can be a setback at times, especially when migrating to new versions of Tcl. But it certainly was not a showstopper since source code was freely available. SID was started with version 7.3 and now runs under 8.0.4. If we were to embark on another project of this scale today, it's our opinion that we would use Tcl again as opposed to the current trend towards JAVA.

Acknowledgements

We thank the open source community, the various tcl extension authors and Scriptics, who laid the foundations on which we could build a successful project of this magnitude.

References

[Ais99] Applied Information Systems home page http://www.ais.com Valid as of 09/01/1999.

[Angel98] Angelovich, Kenny and Sarachan, "NBCs Genesis Broadcast Automation System: From Prototype to Production", *Proc. Sixth Annual Tcl/Tk Conference*, pp. 1-9, San Diego, Calif.:USENIX, 1998.

[DeCl97] De Clarke, "Dashboard: A Knowledge-Based Real-Time Control Panel", *Proc. Fifth Annual Tcl/Tk Workshop*, pp. 9-18, Boston, Mass.:USENIX, 1997.

[Hugh99] The mSQL Home Page, http://www.hughes.com.au Valid as of 09/01/1999.

[PL99] The PLPlot Home Page, http://emma.la.asu.edu/plplot Valid as of 09/01/1999.

[RL98] Reekie H.J. and Lee E.A, "The Tycho Slate: Complex Drawing and Editing in Tcl/Tk", *Proc. Sixth Annual Tcl/Tk Conference*, pp. 37-46, San Diego, Calif.:USENIX, 1998.

[WeFr97] Webster T. and Francis A., "Tcl/Tk for Dummies", pp. 324, IDG Books, 1997.

[Wolf98] Hans K.Wolf, "Java, CORBA, and Architecture", *Component Strategies*, September 1998, pp. 58-64.

Proxy Tk: A Java applet user interface toolkit for Tcl

Mark Roseman
TeamWave Software Ltd.
roseman@teamwave.com

ABSTRACT

Proxy Tk allows a Tcl program to provide a highly interactive web-browser user interface, without requiring the end user to download additional software. It uses a thin client design, where a server-side Tcl application communicates with a very small generic Java applet running in the browser, sending it commands to create and modify widgets, and receiving events from the user. A Tk-like layer encapsulates the communication protocols to provide a familiar programming paradigm, and allow easy porting of existing code.

Introduction

For some applications, running in a user's web browser rather than as a conventional workstation application is important. Existing Tcl solutions support building a wide range of web-based applications, but do not address interactive user interfaces in situations where downloading extra software may be problematic.

This paper describes Proxy Tk, an architecture that allows very interactive user interfaces to be developed in Tcl and run within a web browser, without the need for the end user to download additional software.

It uses a thin client design, where the main Tcl application, rather than running on the user's workstation, runs on the server. It directs a small Java applet running in the user's web browser to provide its user interface. Because this applet is generic, it supports a range of different applications, and can be extended for specific cases. This approach leverages the high-level, rapid development, and easy extension capabilities of Tcl, while still delivering a web user interface through the Java applet.

We begin the paper by examining the demand for web based software, using our own TeamWave Workplace application as an example. After considering several alternative Tcl development approaches, we provide an overview of our solution, and then proceed to detail its two key features, the communication protocols and the Tcl interface. We then examine how we used Proxy Tk to migrate our own software from a standalone to a web based application, and discuss issues that can arise using this approach in various other applications.

Though not a panacea, Proxy Tk solves an important part of the problem of delivering web applications using Tcl.

Web-Based Software

For many application areas, there is a strong preference for software that is delivered to users via their web browser, rather than as a traditional application installed on their own workstation. Arguments for web-based software vary, grounded both in reality and perception.

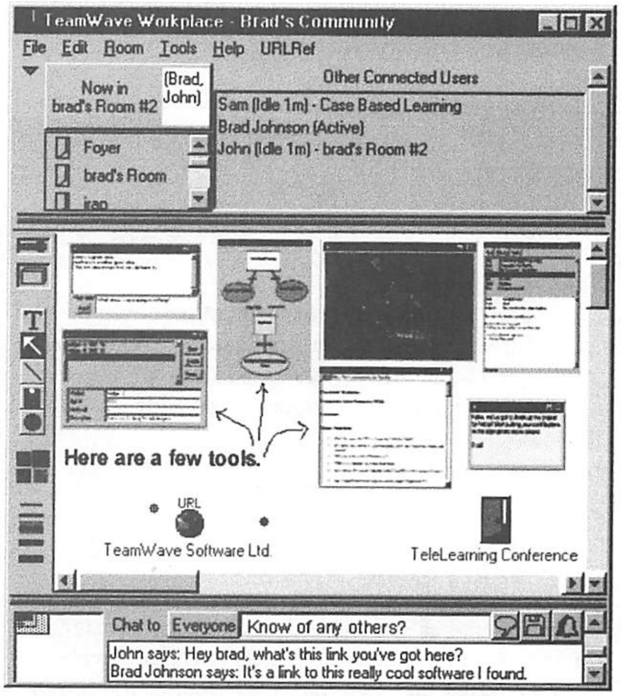
For example, one-time or sporadic tasks may not justify the time, effort, resources and security checks needed to install new software. Downloading new applications or installing system add-ons may be too complicated for novice users [4]. Many people (wrongly!) argue since users already know how to use browsers, web-based applications will have no learning curve. These various concerns often end up encoded in organizational policies, making it even more difficult to adopt software that is not web-based.

While HTML or Javascript solutions are alternatives for many form based applications, programs requiring greater richness or interactivity must rely on either Java or a browser plugin approach. Unfortunately, we hear many of the same sentiments about downloading new applications expressed about plugins (except of course those bundled with the browsers).

TeamWave Workplace

As one example of the demand for web-based software, consider our own TeamWave Workplace application. One or more users running a client application connect to a server, where they share data and communicate with each other, using chat, whiteboards, and other tools. The user interface is shown in Figure 1.

Both the client and server are large programs, written in a mix of C/C++ and Tcl/Tk. The client is particularly complex, containing several different Tcl interpreters, one for each of the tools. The architecture of an earlier prototype of the system is described in [6].



Sam (Idle 1m) - Case Based Learning
Brad Johnson (Active)
John (Idle 1m) - brad's Room #2

Know of any others?

John says: Hey brad, what's this link you've got here?
Brad Johnson says: It's a link to this really cool software I found.

Figure 1. TeamWave Workplace user interface.

While many users were fine with downloading and installing our client (completely "wrapped" and a simple install), we continued to receive requests for a web version. Groups running "virtual communities" needed to lower the barriers for new users to join and to use the community frequently. In practice, users would not make the extra effort to download the software unless they had a very pressing need. In some situations (e.g. Internet cafes, labs), installing software to use the community was impractical. For some of our key markets, a web-browser version was essential.

A Java version seemed the right thing to do, but would involve developing an entirely new code base in parallel with the existing code base of our standalone version. The prospect of abandoning scripting languages for our application in favor of a systems language like Java was also not appealing. We had also just released a Tcl SDK for developers to extend the application, and could see no way to allow this from the Java side.

Finally, we were not thrilled by the prospect of trying to develop a large Java applet, when so many who had attempted such projects had failed. What we really needed was a solution that would preserve our code base as well as the robustness and rapid, higher-level development we were accustomed to with Tcl, yet still run as a Java applet.

Existing Approaches

Tcl developers have several excellent options to them for developing web-based applications, each suited to different types of applications.

Microscripting. Using tclHttpd or another Tcl-savvy web server, small Tcl scripts can be executed on the fly and their results embedded in the HTML page [7]. This solution is restricted by what is possible to deliver in HTML, and therefore cannot be used to deliver a sufficiently rich and interactive user interface.

CGI. Using a library such as cgi.tcl, running against any web server, Tcl developers can easily write CGI scripts to do a variety of forms processing [3]. Again, the user interface presented by CGI scripts is limited to what can be provided by HTML.

Jacl. By providing an implementation of Tcl written entirely in Java, this approach has promise for delivering more interactive user interfaces [1]. However, the current version is both incomplete and cannot run as an applet within a web browser.

VNC. This program [5] (and similar software) takes a Unix X11 desktop and sends it to a remote viewer, similar to how the X11 protocol can work over a network. While a Java applet viewer is available, this solution works with only the entire desktop, not a single window. Multiple instances of the application cannot be run off the same machine; multiple users all share the same desktop display. Further, because it operates at a very low level, high bandwidth and low latency networks are essential.

Tcl/Tk Plugin. The Sun Tcl/Tk Plugin allows running full Tcl/Tk scripts safely in a web browser [2]. Based on the standard Tcl/Tk code base, the plugin is capable of very sophisticated user interfaces. However, because the plugin is not bundled as a standard part of existing browsers, it requires an extra download and install step, which as we have seen may be an obstacle for some applications. Even if the plugin were bundled with existing browsers, a custom plugin would be needed for applications requiring any C extensions.

In this paper, we consider a new approach, capable of delivering very interactive web-based interfaces, yet not requiring users to download additional software.

Architecture Overview

We are basing our approach on a "thin client" design. In a conventional application architecture, the code for the application runs entirely on the user's workstation, using the workstation's own windowing system to present its user interface. This is known as a "thick client" design.

In the "thin client" design we are presenting here, rather than a single component running on the user's workstation, the system consists of two software components, the "thick" server and the "thin" client. The server is a Tcl program running on a network server, and implements the bulk of the application. This is the *same* code that would normally run on the user's workstation in a thick client design — we are now migrating this code to instead run on a network server. This Tcl application communicates with a thin client, which is a generic Java applet, running in the end user's web browser. This thin client provides the user interface for the application.

Though our application is inherently network oriented, in any situation where web browser solutions are called for, servers must exist, making this architecture viable. We'll consider in our case study later in the paper how to apply this architecture to applications that in their original form use a client-server architecture, where the main application running on the user's workstation talks to a server program on a remote machine. But again, in the normal case, we're talking about web-enabling applications that used to run entirely on the user's workstation, with no server component.

The applet doesn't know about our specific application, but is able to respond to simple commands sent from the Tcl server program. These commands direct the applet to create its user interface, and to communicate user interactions back to the server. The Java applet then consists solely of a general purpose "widget server" used by the server-side Tcl program to build the complete user interface at run-time.

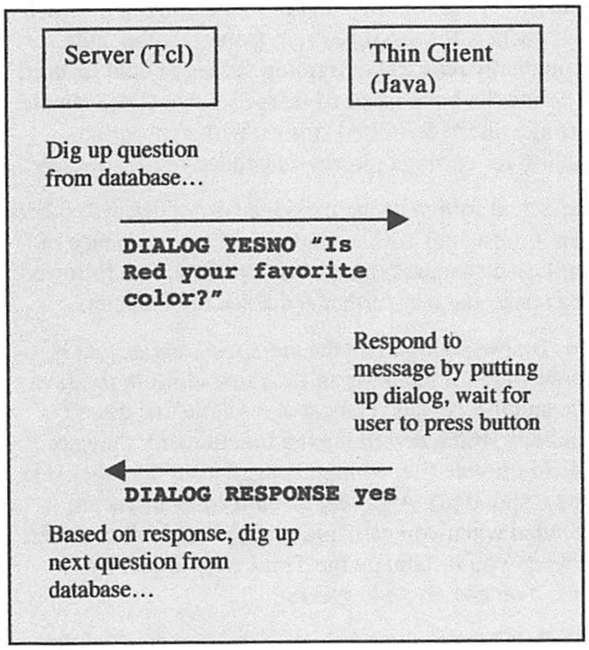

Server (Tcl)
Thin Client (Java)
Dig up question from database...
DIALOG YESNO "Is Red your favorite color?"
Respond to message by putting up dialog, wait for user to press button
DIALOG RESPONSE yes
Based on response, dig up next question from database...

Figure 2. Example of thin client interaction.

As an illustration, consider an application that asks a series of questions of the user, the next question based on the answer to previous ones. Using our architecture, the Tcl server application will know about the different questions and the logic to chose between them. All our Java thin client needs to know is how to ask a question of a user and send the response back to the server. A simple exchange is illustrated in Figure 2.

This dialog example is still fairly high level; the thin client needs considerable knowledge about how to ask a question of the user, even if it doesn't know about the specific questions to ask. What we'll need to make the thin client more general is to work at a lower level, more akin to individual Tk widgets.

Consider what could be built if your Tcl program could communicate to the Java thin client using something like a Tk canvas widget. It could instruct the applet to create various line, shapes, text and other items. The thin client could respond to mouse events by sending messages back to your Tcl program, which you handle in a way specific to your application. It is at this level of granularity that we've actually constructed this architecture.

Design Goals

When designing this system, we had several goals in mind, both reflecting our particular requirements and also what we perceived as the potentially wide-ranging uses of this thin client approach.

Run in current browsers. Because we actually wanted to deploy this software, we needed to balance our desire to use the latest and greatest Java features in the applet against what people were actually running. We chose to target browsers supporting JDK 1.1.

Simplify the thin client. We wanted to design the system, including the network protocol, to make the Java thin client side as simple as possible. Keeping less code in the applet not only makes for a faster download, but allows us to leverage C code extensions on the server for performance and more functionality.

Don't assume Java. Where possible, we wanted to design the protocol so that it was more generic, and didn't assume Java on the thin client end. This would let us consider other implementations, such as e.g. ActiveX on Windows, or design thin clients for platforms that may not support Java well, or at all, e.g. PalmPilot. As such, not only should the protocol syntax not rely on Java-specific tricks (e.g. serialized objects), but the message contents should minimize assumptions about how the Java client would be implemented.

Easy Tcl interface. We didn't want developers to have to interact with communication protocols directly, at least not unless they were adding new capabilities to the toolkit (analogous to coding a new widget in C). The natural choice is a Tk-like interface, where a "proxy widget" on the server interacts with the thin client using the communication protocols. In keeping with the goal of simplifying the thin client, advanced features (e.g. canvas tags) would normally be implemented strictly in these server-side proxies; the thin client would know only about object ids.

Extensible. The protocol should be extensible, so that new functionality can be added if required for particular applications (e.g. more features, better performance). The Tcl server application should also be able to determine if extensions are present on the thin client, and to take advantage of them if so.

Communication Protocols

This section describes the lower level communications protocols upon which the Proxy Tk architecture has been built. We'll describe the actual communications mechanisms by stepping through an application of how it would be actually used.

In our example, we are running our Tcl server application on the machine www.company.com. This machine is also running a web server; because of Java security restrictions, our applet will only be able to connect to servers on the same machine as it is retrieving web pages from.

Making Connections

A user on another machine uses their web browser to open up the web page at this URL:

http://www.company.com/myapp.html

This page contains the following HTML:

```
<html>
<body>
<applet code="ThinClient.class"
        width="400"
        height="400">
<param  name="host"
        value="www.company.com">
<param  name="port"
        value="9900">
</applet>
</body>
</html>
```

When this web page is loaded, it will run the applet stored in ThinClient.class, passing as parameters the host and port of our Tcl server application. On starting, the applet will make a socket connection to this server.

The Tcl server application is written something like the following. It sets up a listening socket, and waits for connections from a thin client, When it gets one, the server creates a new Tcl interpreter for that particular thin client. It then loads a Tcl package called "proxy" into the new interpreter, which provides routines to send and receive messages from the thin client, defines proxy widgets, etc. It then transfers the socket to the new interpreter. Finally, it sources the Tcl program containing the code for our application. That program will then communicate with the thin client.

Using multiple interpreters, one for each client, removes the possibilities for namespace conflicts between client proxies, and also opens up the possibility of using one thread per thin client, for performance. Though not a fundamental characteristic of our architecture, using multiple interpreters did prove to be a useful design choice in our case.

```
# Main server application code
socket -server acceptCmd 9900

proc acceptCmd {sock addr port} {
   set interp [interp create]
   interp transfer "" $sock $interp
   $interp eval package require proxy
   $interp eval proxy::init $sock \
      $addr $port
   $interp eval source myapp.tcl
}
```

Messages

Both the Tcl server-side and the Java thin client watch their sockets for *messages* sent from the other side. Though any reasonable framing strategy could be used to define the boundaries of messages, we chose simple carriage return delimited strings, with appropriate quoting for carriage returns embedded in the message.

The actual format of the messages is not defined, other than it must start with a single word (i.e. sequence of non-space characters), and may optionally be followed by a space and any further sequence of characters.

This first word signifies the *message handler* that is responsible for handling this command. Both the Java side and the Tcl side consist of a number of these handlers, which determine the functionality they are able to provide (i.e. what messages from the other side they respond to). A variety of built-in handlers are included when you call "proxy::init" on the Tcl server, or when you instantiate the ThinClient applet from Java. You can also add others.

Handlers have a name (which is the first word of the network messages), a version (for tracking enhancements over time), and a handler proc which responds to the message. On the Tcl side, this is just a Tcl procedure taking a single argument, namely the extra parameters sent along as part of the network message. A handler proc is registered using the "proxy::handler" Tcl command e.g.

```
# handler for FOO version 0.1
proc fooHandler {params} {
  ... interpret params
}

proxy::handler FOO fooHandler 0.1
```

Server side handlers can also be defined in C; a similar API is used in that case. A C handler proc consists of a single function taking both a string parameter (for parameters), and a pointer to an internal structure holding data for that proxy client.

On the Java side, a handler is any class that implements an interface called MessageHandler, containing a single handle( ) method. To add new packages to the applet, the programmer generally creates a new subclass of the ThinClient class, and calls its registerHandler method to add each new package. This is illustrated below.

```
class FooHandler
       implements MessageHandler {
  public void handle(String params) {
    ... interpret params
  }
}

public class MyClientApplet
      extends ThinClient {
  public void registerHandlers() {
    super.registerHandlers();
    m_client.registerHandler("FOO",
        "0.1", new FooHandler());
  }
}
```

Determining Capabilities

One of the requirements in this system is that the application logic contained in the server needs to know what packages the thin client implements, so that it knows what facilities for display are available to it. This might be used for example, so that different applets (with more features) can be run on newer browsers that support more recent versions of the JDK.

When the thin client first connects up, it sends a message to the server telling it what handlers (and what versions of each handler) it has available. The message might look something like this:

```
HANDLERS CANVAS 1.0 ENTRY 0.5...
```

On the Tcl side, the first word of the command is stripped out ("HANDLERS") and the remainder of the command sent to the handler for the (built-in) handler named HANDLERS. This handler stores the list, and makes it available to the Tcl application via the "proxy::remotehandlers" command. With no parameters, this command returns a list of all known handler names, while passing it the name of a handler returns the version of that particular handler.

The message format above would be fairly typical for most handlers, because it is easy to parse. But again, it is worth stressing that this system assumes nothing about the message format after the first word. The routines that read in messages just look for the first word in their list of registered handlers, and send the rest of the message on to the appropriate handler.

Defining Widgets

Not surprisingly, each widget that is provided by the thin client implements a single handler. For example, the thin client's button widget is accessed through messages starting with the word "BUTTON". Most widgets have some elements in common, such as an id number to refer to them, a fairly uniform way of formatting the underlying messages (e.g. "handler id operation options...") and similar operations that are available (e.g. create, delete, set configuration options).

To use the button widget as an example, the server would send the following two messages to the thin client to create a button (having id 25) and set its label:

```
BUTTON 25 new
BUTTON 25 set label Push Me!
```

On the Java side, a singleton instance of a ButtonHandler class would receive and interpret the messages. The new message would cause the handler to create a new instance of a ButtonWidget (a wrapper around AWT's Button class) and store it in a table of all known widgets, indexed by its id. The set message would cause the handler to look up the ButtonWidget in the table, and call its 'set' method, which would modify the properties of the underlying AWT button.

Notice that the Java side has not received any instructions as to what to do when the user presses the button (e.g. the equivalent of the "-command" option in Tk). The thin client has no knowledge of how the application should deal with events; anytime the button is pressed, it will simply send a message back to the server, which can handle it as it chooses:

```
BUTTON 25 buttonPress
```

The application code running in the server will have to remember that button 25 corresponds to the "Push Me!" button, and know what to do when it is pressed. While it would be possible to develop programs at this low level that use the Java thin client, clearly a higher level interface that hides all these protocol details would be beneficial. We'll cover just such a programming interface in the next section.

Tcl Interface

The previous section outlined the communications protocol which defines the low-level interface between the server-side application and the Java thin client display. The specific operations supported by the protocol determine what functionality in the thin client is available to the application for its use.

However, its clear that working at such a low level — formatting and parsing network messages, caching proxy widget state information, and keeping track of widgets by id number — are not exactly conducive to the high level programming practices we've become used to with tools like Tcl and Tk!

Therefore, we created a Tcl interface that encapsulates all the low level protocols, caches widget information, and does all the housekeeping one would expect.

Not surprisingly, the design of this Tcl interface bears more than a passing similarity to Tk. A widget creation command (e.g. button) is used to create new widgets, their object command (e.g. .proxy.b) is used to refer to them and manipulate them, configuration options can be set and queried, event bindings are available, etc. Table 1 provides a summary of what subset of Tk has been implemented in the current version of Proxy Tk.

Because the API is mostly just a subset of the API for Tk, it's a fairly straightforward manner to either port existing code, or have a code base that runs well under both Tk and Proxy Tk. We'll discuss this in a bit more detail in the next section. We'll also touch on some of the differences that arise because the actual user interface is running remotely from the application, and how these can be addressed.

The implementation of the proxy widgets on the Tcl side is also straightforward. Each widget typically requires three things. A message handler interprets messages it receives from the thin client. A widget creation command creates both a new Tcl command to refer to the widget, and a cache for any data associated with the widget, Finally, a widget object command is used to set configuration options and perform other operations. Each of these will format and send messages over the network to the Java applet.

Command	**Options**
button	-text, -command, -font
entry	get, insert, -width, -show, -font
menu	post, add/insert; command, separator, radiobutton, entryconfigure; –label –command –state
text	insert, delete, get, -width –height –font –color –state –background
listbox	insert, delete, get, see, curselection,–font
canvas	create line, rectangle, oval, text, window, image; bbox, canvasx, canvasy, delete, coords, itemconfigure, itemcget, bind, find withtag / overlapping / enclosed, type, gettags, raise, lower, –width, –height, –text, –background, –scrollregion; –tags, –fill, –outline,–font, –anchor, –window
grid	–in, –column, –row, –columnspan, –rowspan, –sticky
bind	which events depends on widget
destroy	
focus	-force
winfo	width, height, exists

Table 1. Summary of current Proxy Tk Tcl interface.

Example

To give a quick example of the correspondence between the Tcl interface and the underlying network protocol, consider the freehand drawing program shown in Figure 3. Buttons at the top control the color of the

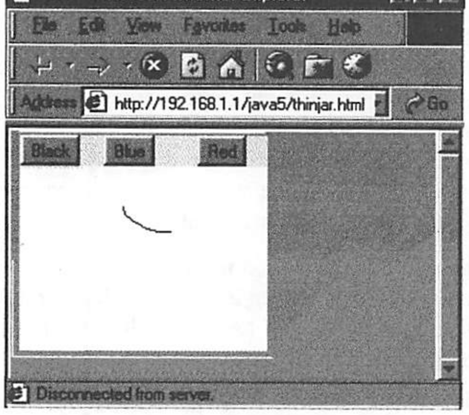

test1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://192.168.1.1/java5/thinjar.html
Go
Black
Blue
Red
Disconnected from server.

Figure 3. Simple Proxy Tk drawing program.

drawing. Clicking and dragging in the canvas draw in the selected color. Code is below.

```
# simple freehand drawing program

# root window
set w .proxy

# import commands into the root namespace
namespace import proxy::*

# create buttons at the top for choosing
grid [button $w.black –text Black \
   -command "set color black"] \
   –column 0 –row 0
grid [button $w.blue –text Blue \
   -command "set color blue"] \
   –column 1 –row 0
grid [button $w.red –text Red \
   -command "set color red"] \
   –column 2 –row 0
set color black

# canvas for drawing
grid [canvas $w.c –background white] \
   –column 0 –columnspan 3 –row 1 \
   -sticky nwes
bind $w.c <1> "set x %x; set y %y"
bind $w.c <B1-Motion> {
   $w.c create line $x $y %x %y \
      –fill $color
   set x %x; set y %y
}
```

When this program first starts up, the following network messages are sent from the thin client to create the user interface (comments in italics).

```
# create each button; the grid is handled
# by the root window (id=1), which is a
# canvas widget in Proxy Tk
BUTTON 2 new
BUTTON 2 set text Black
CANVAS 1 grid activate
CANVAS 1 grid add 2 column=0 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

BUTTON 3 new
BUTTON 3 set text Blue
CANVAS 1 grid add 3 column=1 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

BUTTON 4 new
BUTTON 4 set text Red
CANVAS 1 grid add 4 column=2 row=0
  columnspan=1 rowspan=1 fill=none
  anchor=center

CANVAS 5 new
CANVAS 5 set background ffffff
CANVAS 1 grid add 5 column=0 row=1
  columnspan=3 rowspan=2 fill=nwes
  anchor=center
```

If the user clicks the red color button, the following message is sent back to the server.

```
BUTTON 4 buttonPress
```

Finally, as the user clicks and draws on the canvas, messages like the following sequence are sent from the thin client, followed by a message back from the server, telling the client to actually draw the line on the display.

```
# mouse down and drag event
CANVAS 5 mousePressed x=26 y=32
CANVAS 5 mouseDragged x=30 y=34

# server sends create a line item, id 6
CANVAS 5 create line 6 26 32 30 34
CANVAS 5 itemset 6 color ff0000

# event from client…
CANVAS 5 mouseDragged x=32 y=36

# … and server tells us to draw, etc.
CANVAS 5 create line 7 30 34 32 36
CANVAS 5 itemset 7 color ff0000
```

Proxy Tk in Practice

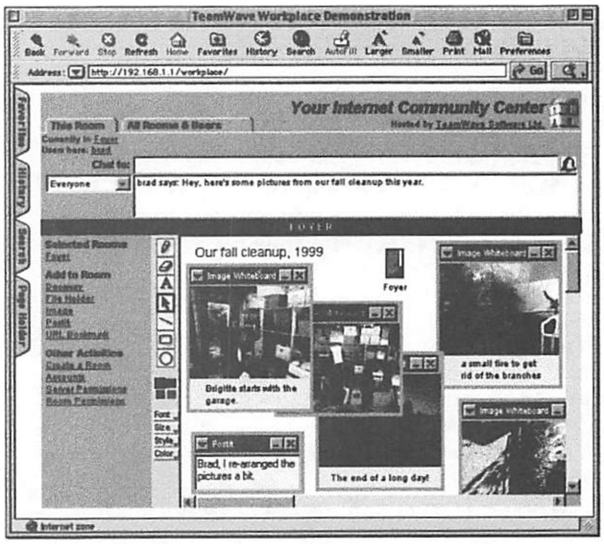
We used Proxy Tk to port our TeamWave Workplace software described earlier to run in a web browser. In this section, we'll describe our experiences with doing the port, and identify some of the issues that arose as a result of using this thin client architecture, and the solutions we found.

In describing how we used Proxy Tk, we can't claim to provide a perfectly objective evaluation. The Proxy Tk infrastructure was developed in conjunction with the port of Workplace, and the subset of widget features available to date are those that we required. Other projects may need to implement different capabilities depending on their needs. As well, we took the opportunity to redesign large parts of the user interface, partly to make the application look more "web-like", but also to improve the existing interface. As such, we can't make many useful direct comparisons between the web version and the original standalone client.

Overview of Workplace Port

The new web-based user interface for TeamWave Workplace is shown in Figure 4. Much of the user interface "around" the whiteboard has been changed from the original client, though the bulk of the program running "inside" the whiteboard remains similar.

First, some general comments are in order. The most important thing is that we were able to successfully use Proxy Tk to deliver a web-based version of our application. We were able to do this very quickly, and



brad says: Hey, here's some pictures from our fall cleanup this year.

Our fall cleanup, 1999

Brigitte starts with the garage.

a small fire to get rid of the branches

Brad, I re-arranged the pictures a bit.

The end of a long day!

Figure 4. Web-based TeamWave Workplace.

were able to reuse the vast majority of our existing code (which still is used by the standalone Tk client). While many issues arose, we have been able to successfully deal with them. The applet that is downloaded to the browser is a tiny 75k in size, making it no larger than many images on web sites. From our point of view, we are very satisfied with this approach, which lets us continue to enjoy Tcl development practices while delivering web-based software.

We had to modify approximately 5-10% of the existing Tcl/Tk code to make it work with Proxy Tk. The changes were needed to address different naming for widgets, and to work around features that had not — or could not —be implemented using Proxy Tk.

In what follows, we'll describe some of the problems and solutions required by the thin client approach used in Proxy Tk.

The first problem was quite specific to our application. We had used multiple Tcl interpreters even within each client (e.g. for each of the tools in the whiteboard), and a fairly complex scheme whereby Tk was shared among them. When we added multiple interpreter support to Proxy Tk, we chose a simpler approach to sharing widgets between interpreters, which resulted in the names of the widgets being different. We needed to restructure the code to figure out the topmost widget name once (rather than just assuming it was "."), and base other widget names from that. This is a convention we should have had to begin with.

Network Latency

In the drawing program example, new lines are seen on the whiteboard only after a round trip of messages from the thin client to the server (to inform the server of a mouse move) and back (to draw the line). On any kind of local area network, such as a corporate intranet, this presents no problems whatsoever. On high latency networks, this can lead to very long lags in feedback to user actions, though in practice even on long-haul Internet links we have not found very severe problems. Most operations, such as selecting tools in our tool palette do not suffer from slower feedback.

This effect is minimized because most widget updates (e.g. when text is entered into an entry widget) are handled purely on the client side by the native Java widgets. In these cases, there is no interaction with the server, so no delay exists. One clear advantage of this approach over protocols like X11 or VNC is that screen refresh and most event handling is done locally on the thin client, and not sent over the network. Perceived performance is therefore comparable to running a local application.

For situations like drawing in the canvas, we have designed but not yet implemented a solution. The server could provide the thin client with a "response template" for a widget. This would be similar to a Tk binding (including percent substitutions). However, rather than it being a Tcl command, it would instead be a network message, such as the thin client might actually receive from the server. On a mouse drag in the affected widget, the thin client would use the template to create the message it would normally receive, and insert it into its own network queue. In this way, feedback is received without the need for a network round trip.

One could get arbitrarily complex with substitutions for response templates (in fact designing a complete scripting language for them!), but analyzing a number of situations where quicker performance may be warranted showed the simple case would handle most of them quite well. Essentially this technique allows us to design in application-specific performance enhancements when required.

Current Widget Contents

Unlike with standard Tk, using proxy widgets means that the current state of the widget, as visible by the user, and the state as visible by the application running on the server, may be quite different. Network messages must be sent between them to synchronize them. This has a number of implications.

First, when widgets are changed by the user, the changes must be sent back to the server. For many widgets, e.g. checkbuttons, this can easily be done every time the widget is changed. For others, e.g. entries or text widgets, a user typing may generate a lot of network traffic; much of this may not be needed, as the application may only care about the contents of the widget after all data is entered, e.g. in a dialog.

We have added a new boolean configuration option, –immediatefeedback to entry and text widgets for this purpose. When true, changes are sent back on every keystroke. When false, changes are not sent back on every change, but only when directed to do so by another widget. As an example, buttons are set up so that before informing the server that they have been pressed, they first instruct all of their sibling widgets (i.e. those having the same parent window) to send their current value back to the server. This has proved for example an effective solution for dialog boxes, where hitting the "Okay" button ensures all widgets in the dialog box first report their current state.

Physical Size of Objects

A related issue is that the server generally does not know the physical size of widgets or objects in the thin client, particularly text objects (which rely on font metrics unknown to the server application). We'll take two examples where this became an issue.

In the first example, we wanted to display selection handles around text objects that have been selected on the whiteboard. In Tk, we do this by using the canvas' "bbox" command to determine the current bounding box of the item. With Proxy Tk, we've made the thin client send back the current bounding box of any changed text items as soon as it knows them (i.e. after a redraw). As it turns out, we never need to change the contents of the text, and immediately select the item, so we can get away with this delay. Otherwise, we'd have to wait for the bounding box to be returned before we could select the text. Note that our caching mechanism has to be clever as well; when we move the object, we can immediately update the cached bounding box, because we know just moving the object will not change its size.

A more problematic situation came up in implementing the URL-like strings shown in the user interface along the top and left. In general, these consist of one or more regions of black and blue text abutted together, with the blue parts being clickable, having status line feedback, etc.

On a regular Tk canvas, we would have created a text item for each segment, determining its starting point by looking at the bounding box of the previous segment. In Proxy Tk, we can't do this, unless we're willing to wait the round trip message after each segment. Instead, because these URL-like strings are pervasive in the new interface, we decided to implement the behavior directly in the Proxy Tk core. Each string is created as a single text item, and we've added a new configuration option called –highlightranges. The value of this option is a list specifying what ranges of the string are highlighted, what command to invoke when they are clicked on, and a status line message for each. All processing of this option's associated behavior is handled locally by the thin client.

Client-Server Architectures

Our description of Proxy Tk has assumed that the program we're web-enabling consists of a single application that runs on the user's workstation. Workplace actually uses a client-server architecture, where the client application on the user's workstation communicates with a server program.

We could have run the Proxy Tk-based version on the server machine as a separate process, and have it make a socket connection to the actual server, running on the same machine. Instead, we took the small bit of code that accepts connections from the Java proxy and spawns new interpreters, and placed that in our existing server. So our server started up new copies of the client application in its own process, as usual with a new Tcl interpreter.

Instead of using actual socket connections (which would have worked), we developed a new Tcl channel type, loosely based on Andreas Kupries' memory channel. Our version was used to connect two Tcl interpreters in the same process together. Data written by one interpreter could be read by the other.

Extensibility, Scalability and Portability

One problem with this approach, and Java in general, is that if you really need some functionality that is not available directly through Java applets, you're pretty much out of luck — no dropping down to native code.

On the other hand, running in a web browser does give us access to all kinds of facilities. Displaying a web page to the user for example is implemented as a five line Proxy Tk message handler.

To allow files to be downloaded to the user's hard drive (which can't be done directly from Java), we added a very simple httpd server to our server application. We instruct the thin client to connect to this server to download files. This server looks for URL's with a particular format (e.g. starting with /files/) and copies the file from our own file store. Images are also handled in this manner; any URL's starting with /image/ are mapped to an internal table. Java provides the built-in routines to create an image from a URL.

Uploading files to the server (again not allowed in Java) proved trickier. The thin client is instructed to connect to our built-in httpd server, at a URL that generates an HTML form containing a file upload field. The user can then use this mechanism to choose a file from their local machine, and submit the form, which uploads the file to our httpd server. The file is then finally moved into our server file store for use by other parts of the program. Note that both file upload and download involve explicit confirmation with the user on the workstation side, and server-side storage is within a file store controlled by our own server software, not the server's general file system. As such, this does not introduce new security issues.

Scalability is one potential issue with thin client architectures, as now the server is responsible for the majority of processing that would otherwise be handled by the client's workstation. How many simultaneously connected clients a server will support will depend entirely on the application. In our measurements, the Proxy Tk layer itself adds little overhead. For our own moderately complex application, we found the server could scale quite well. We had to be careful not to have any one operation process for too long without reentering the event loop. Doing so would prevent other clients from obtaining processing time. Breaking up long operations into several pieces is a common technique needed by many such event-driven, cooperative multitasking systems.

Finally, a word about portability is in order. Its no secret that despite all the hype, because of all the bugs in the various Java Virtual Machines, Java remains a "write once, debug everywhere" language. At present, our applet runs on only a small number of JVMs, though we are in the process of making it more portable. Not surprisingly, we've found that making our 75k applet work on different JVMs is an order of magnitude easier than doing so for a thick client applet. To work around a bug in Java's Button widget for example, requires changing one piece of our code, not one piece for every button created by the application.

Conclusions

We've described Proxy Tk, which allows Tcl programmers to deliver web-based applications using a thin client architecture. Using an API that offers a subset of Tk functionality, a high-level Tcl program can control a 75k Java applet to implement its user interface. This technique extends the range of solutions available for delivering web-based applications using Tcl, and is particularly suited for highly interactive applications that must run in situations where downloading new software would be prohibitive.

Our experience in using Proxy Tk to develop a web-based interface for TeamWave Workplace suggests that existing Tcl/Tk code can be ported with fairly minimal changes, and that problems encountered because of the architecture are surmountable. We believe that using the techniques described here would be an effective means for developing web-based versions of a wide range of Tcl applications.

Acknowledgements

Thanks to Leo Pelland, who wrote the Java side of the prototype that led to this design and implementation. Feedback on earlier drafts of this paper was provided by Leo Pelland, Ted O'Grady, and Larry Virden.

References

1. Lam, I. and Smith, B. *Jacl: A Tcl Implementation in Java*. Proceedings of the Fifth Annual Tcl/Tk Workshop. July, 1997.
2. Levy, J. *A Tk Netscape Plugin*. Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
3. Libes, D. *Writing CGI Scripts in Tcl*. Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
4. Nielsen, J. *The increasing conservatism of web users*. Alertbox. March 22, 1998. http://www.useit.com/alertbox/980322.html
5. Richardson, T., Stafford-Fraser Q., Wood, K. and Hopper, A. *Virtual Network Computing*. IEEE Internet Computing 2(1), 1998. Also see http://www.uk.research.att.com/vnc/
6. Roseman, M. *Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application*. Proceedings of the Fourth Annual Tcl/Tk Workshop. July, 1996.
7. Welch, B. and Uhler, S. *Web Enabling Applications*. Proceedings of the Fifth Annual Tcl/Tk Workshop. July, 1997.

The TclHttpd Web Server

Brent Welch <welch@scriptics.com>
Scriptics Corporation

Abstract

This paper describes TclHttpd, a web server built entirely in Tcl. The web server can be used as a stand-alone server or it can be embedded into applications to web-enable them. TclHttpd provides a Tcl+HTML template facility that is useful for maintaining site-wide look and feel, and an application-direct URL that invokes a Tcl procedure in an application. This paper describes the architecture of the application and relates our experience using the system to host www.scriptics.com.

Introduction

TclHttpd started out as about 175 lines of Tcl that could serve up HTML pages and images. The Tcl `socket` and I/O commands make this easy. Of course, there are lots of features in web servers like Apache or Netscape that were not present in the first prototype. Steve Uhler took my prototype, refined the HTTP handling, and aimed to keep the basic server under 250 lines. I went the other direction, setting up a modular architecture, adding in features found in other web servers, and adding some interesting ways to connect TclHttpd to Tcl applications.

Today TclHttpd is used both as a general-purpose Web server, and as a framework for building server applications. It implements `www.scriptics.com`, including the Tcl Resource Center and Scriptics' electronic commerce facilities. It is also built into several commercial applications such as license servers and mail spam filters.

Integrating with TclHttpd

TclHttpd is interesting because, as a Tcl script, it is easy to add to your application. Suddenly your application has an interface that is accessible to Web browsers in your company's intranet or the global Internet. The Web server provides several ways you can connect it to your application:

- *Static pages*. As a "normal" web server, you can serve static documents that describe your application.
- *Domain handlers*. You can arrange for all URL requests in a section of your web site to be handled by your application. This is a very general interface where you interpret what the URL means and what sort of pages to return to each request. For example, `http://www.scriptics.com/resource` is implemented this way. The URL past `/resource` selects an index in a simple database, and the server returns a page describing the pages under that index.
- *Application-Direct URLs*. This is a domain handler that maps URLs onto Tcl procedures. The form query data that is part of the HTTP GET or POST request is automatically mapped onto the parameters of the application-direct procedure. The procedure simply computes the page as its return value. This is an elegant and efficient alternative to the CGI interface. For example, in TclHttpd the URLs under `/status` report various statistics about the web server's operation.
- *Document handlers*. You can define a Tcl procedure that handles all files of a particular type. For example, the server has a handler for CGI scripts, HTML files, image maps, and HTML+Tcl template files.
- *HTML+Tcl Templates*. These are web pages that mix Tcl and HTML markup. The server replaces the Tcl using the `subst` command and returns the result. The server can cache the result in a regular HTML file to avoid the overhead of template processing on future requests. Templates are a great way to maintain common look and feel to a family of web pages, as well as to implement more advanced dynamic HTML features like self-checking forms.

TclHttpd Architecture

Figure 1 shows the basic components of the server. At the core is the `Httpd` module, which implements the server side of the `HTTP` protocol. This module manages network requests, dispatches them to the `Url` module, and provides routines used to return the results to requests. The `Url` module divides the web site into *domains*, which are subtrees of the URL hierarchy provided by the server. The idea is that different domains may have completely different implementations. For example, the Document domain maps its URLs into files and directories on your hard disk, while the Application-Direct domain maps URLs into Tcl procedure calls within your application. The CGI domain maps URLs onto other programs that compute web pages.

Domain Handlers

You can implement new kinds of domains that provide your own interpretation of a URL. This is the most flexible interface available to extend the web server. You provide a callback that is invoked to handle every request in a domain, or subtree, of the URL hierarchy. The callback interprets the URL, using routines from the `Httpd` module.

Example 1 defines a simple domain that always returns the same page to every request. The domain is registered with the `Url_PrefixInstall` command. The arguments to `Url_PrefixInstall` are the URL prefix and a callback that is called to handle all URLs that match that prefix. In the example, all URLs that have the prefix `/simple` are dispatched to the `SimpleDomain` procedure.

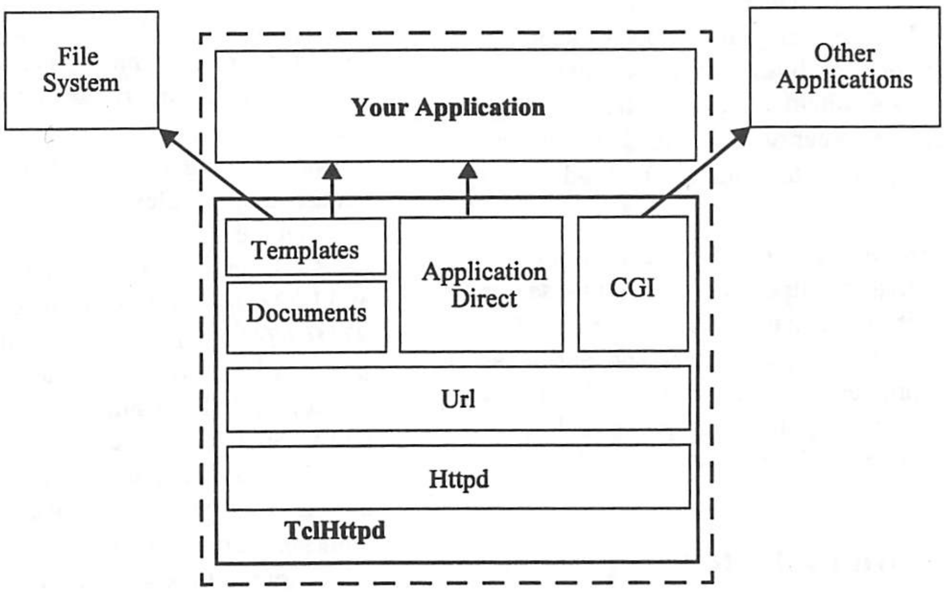


Figure 1 The dotted box represents one application that embeds TclHttpd. Document templates and Application Direct URLs provide direct connections from an HTTP request to your application.

Example 1 A simple URL domain.

```
Url_PrefixInstall /simple [list SimpleDomain /simple]

proc SimpleDomain {prefix sock suffix} {
    upvar #0 Httpd$sock data

    # Generate page header

    set html "<title>A simple page</title>\n"
    append html "<h1>$prefix$suffix</h1>\n"
    append html "<h1>Date and Time</h1>\n"
    append html [clock format [clock seconds]]
    # Display query data

    if {[info exist data(query)]} {
        append html "<h1>Query Data</h1>\n"
        append html "<table>\n"
        foreach {name value} [Url_DecodeQuery $data(query)] {
            append html "<tr><td>$name</td>\n"
            append html "<td>$value</td></tr>\n"
        }
        append html "</table>\n"
    }
    Httpd_ReturnData $sock text/html $html
}
```

The `SimpleDomain` handler illustrates several properties of domain handlers. The `sock` and `suffix` arguments to `SimpleDomain` are appended by `Url_Dispatch` when it invokes the domain handler. The `suffix` parameter is the part of the URL after the prefix. The `prefix` is passed in as part of the callback definition so the domain handler can recreate the complete URL. For example, if the server receives a request for the url `/simple/page`, then the prefix is `/simple`, the suffix is `/page`.

The `sock` parameter is a handle on the socket connection to the remote client. This variable is also used to name a state variable that the `Httpd` module maintains about the connection. The name of the state array is `Httpd$sock`, and `SimpleDomain` uses `upvar` to get a more convenient name for this array (i.e., `data`):

```
upvar #0 Httpd$sock data
```

The only module in the server that uses the socket handle directly is the Httpd module. The rest of the code treats `$sock` as an opaque handle, and uses the `upvar` trick to map that handle into a locally accessible array.

An important element of the state array is the query data, `data(query)`. This is the information that comes from HTML forms. The query data arrives in an encoded format, and the `Url_DecodeQuery` procedure is used to decode the data into a list of names and values.

Finally, once the page has been computed, the `Httpd_ReturnData` procedure is used to return the page to the client. This takes care of the HTTP protocol as well as returning the data. There are three related procedures, `Httpd_ReturnFile`, `Httpd_Error`, and `Httpd_Redirect`.

Application Direct URLs

The Application Direct domain implementation provides the simplest way to extend the web server. It hides the details associated with query data, decoding URL paths, and returning results. All you do is define Tcl procedures that correspond to URLs. Their arguments are automatically matched up to the query data. The Tcl procedures compute a string that is the result data, which is usually HTML. That's all there is to it.

The `Direct_Url` procedure defines a URL prefix and a corresponding Tcl command prefix. Any URL that begins with the URL prefix will be handled by a corresponding Tcl procedure that starts with the Tcl command prefix. This is shown in Example 2:

Example 2 Application Direct URLs

```
Direct_Url /demo Demo

proc Demo {} {
    return "<html><head><title>Demo page</title></head>\n\
        <body><h1>Demo page</h1>\n\
        <a href=/demo/time>What time is it?</a>\n\
        <form action=/demo/echo>\n\
        Data: <input type=text name=data>\n\
        <br>\n\
        <input type=submit name=echo value='Echo Data'>\n\
        </form>\n\
        </body></html>"
}
proc Demo/time {{format "%H:%M:%S"}} {
    return [clock format [clock seconds] -format $format]
}
proc Demo/echo {args} {
    # Compute a page that echos the query data

    set html "<head><title>Echo</title></head>\n"
    append html "<body><table>\n"
    foreach {name value} $args {
        append html "<tr><td>$name</td><td>$value</td></tr>\n"
    }
    append html "</tr></table>\n"
    return $html
}
```

Example 2 defines `/demo` as an Application Direct URL domain that is implemented by procedures that begin with `Demo`. There are just three URLs defined:

```
/demo
/demo/time
/demo/echo
```

The `/demo` page displays a hypertext link to the `/demo/time` page, and a simple form that will be handled by the `/demo/echo` page. This page is static, and so there is just one `return` command in the procedure body.

The `/demo/time` procedure just returns the result of `clock format`. It doesn't even bother adding `<html>`, `<head>`, or `<body>` tags, which you can get away with in today's browsers. A simple result like this is also useful if you are using programs to fetch information via HTTP requests. The `/demo/time` procedure is defined with an optional `format` argument. If a `format` value is present in the query data then it overrides the default value given in the procedure definition.

Using Query Data

The /demo/echo procedure creates a table that shows its query data. Its `args` parameter gets filled in with a name-value list of all query data. You can have named parameters, named parameters with default values, and the `args` parameter in your application-direct URL procedures. The server automatically matches up incoming form values with the procedure declaration. For example, suppose you have an application direct procedure declared like this:

`proc Demo/param { a b {c cdef} args}` *body*

You could create an HTML form that had elements named `a`, `b`, and `c`, and specified /demo/param for the `ACTION` parameter of the `FORM` tag. Or, you could type the following into your browser to embed the query data right into the URL:

```
/demo/param?a=5&b=7&c=red&d=%7ewelch&e=tw
o+words
```

In this case, when your procedure is called, `a` is `5`, `b` is `7`, `c` is `red`, and the `args` parameter becomes a list of:

```
d ~welch e {two words}
```

Returning Other Content Types

The default content type for application direct URLs is `text/html`. You can specify other content types by using a global variable with the same name as your procedure. (Yes, this is a crude way to craft an interface.) Example 3 shows part of the `faces.tcl` file that implements an interface to a database of picons, or personal icons, that is organized by user and domain names. The idea is that the database contains images corresponding to your email correspondents. The `Faces_ByEmail` procedure, which is not shown, looks up an appropriate image file. The application direct procedure is `Faces/byemail`, and it sets the global variable `Faces/byemail` to the correct value based on the filename extension. This value is used for the `Content-Type` header in the result part of the `HTTP` protocol.

Document Types

The Document domain (`doc.tcl`) maps URLs onto files and directories. It provides more ways to extend the server by registering different document type handlers. This occurs in a two step process. First the type of a file is determined by its suffix. The `mime.types` file contains a map from suffixes to `MIME` types such as `text/html` or `image/gif`. This map is controlled by the `Mtype` module in `mtype.tcl`. Second, the server checks for a Tcl procedure with the appropriate name:

`Doc_`*`mimetype`*

The matching procedure, if any, is called to handle the URL request. The procedure should use routines in the `Httpd` module to return data for the request. If there is no matching `Doc_mimetype` procedure, then the default document handler uses `Httpd_ReturnFile` and specifies the Content Type based on the file extension:

```
Httpd_ReturnFile $sock [Mtype $path] $path
```

Example 3 Alternate types for Application Direct URLs.

```
Direct_Url /faces Faces
proc Faces/byemail {email} {
    global Faces/byemail
    set filename [Faces_ByEmail $email]
    set Faces/byemail [Mtype $filename]
    set in [open $filename]
    fconfigure $in -translation binary
    set X [read $in]
    close $in
    return $X
}
```

Example 4 A sample document type handler.

```
# Add this line to mime.types
application/myjunk      .junk

# Define the document handler procedure
#   path is the name of the file on disk
#   suffix is part of the URL after the domain prefix
#   sock is the handle on the client connection

proc Doc_application/myjunk {path suffix sock} {
    upvar #0 Httpd$sock data
    # data(url) is more useful than the suffix parameter.

    # Use the contents of file $path to compute a page
    set contents [somefunc $path]

    # Determine your content type
    set type text/html

    # Return the page
    Httpd_ReturnData $sock $type $data
}
```

You can make up new types to support your application. Example 4 shows the pieces need to create a handler for a fictitious document type `application/myjunk` that is invoked to handle files with the `.junk` suffix. You need to edit the `mime.types` file and add a document handler procedure to the server:

As another example, the HTML+Tcl templates use the `.tml` suffix that is mapped to the `application/x-tcl-template` type. The TclHttpd distribution also includes support for files with a `.snmp` extension that implement a template-based web interface to the Scotty SNMP Tcl extension.

HTML + Tcl Templates

The template system uses HTML pages that embed Tcl commands and Tcl variable references. The server replaces these using the `subst` command and returns the results. The server comes with a general template system, but using `subst` is so easy you could create your own template system. The general template framework has these components:

- Each `.html` file has a corresponding `.tml` template file. This feature is enabled with the `Doc_CheckTemplates` command in the server's configuration file. Normally, the server returns the `.html` file unless the corresponding `.tml` file has been modified more recently. In this case the server processes the template, caches the result in the `.html` file, and returns the result.
- A dynamic template (e.g., a form handler) must be processed each time it is requested. If you put the `Doc_Dynamic` command into your page it turns off the caching of the result in the `.html` page. The server responds to a request for a `.html` page by processing the `.tml` page. Or, you can just reference the `.tml` file directly, in which case the server always processes the template.
- The server creates a `page` global Tcl variable that has context about the page being processed.
- The server initializes the `env` global Tcl variable with similar information, but in the standard way for CGI scripts.
- The server supports per-directory "`.tml`" files that contain Tcl source code. These files are designed to contain procedure definitions and variable settings that are shared among pages. The name of the file is simply "`.tml`", with nothing before the period. The server will source the "`.tml`" files in all directories leading down to the directory containing the template file. The server compares the modify time of these files against the template file and will process the template if these "`.tml`" files are newer than the cached `.html` file. So, by modifying the "`.tml`" file in the root of your URL hierarchy you invalidate all the cached `.html` files.
- The server supports a script library for the procedures called from templates. The `Doc_TemplateLibrary` procedure registers this directory. The server adds the directory to its `auto_path`, which assumes you have a `tclIndex` or `pkgIndex.tcl` file in the directory so the procedures are loaded when needed.

Where to put your Tcl Code

There are three places you can put the code of your application: directly in your template pages, in the per-directory "`.tml`" files, or in the library directory. The advantage of putting procedure definitions in the library is that they are defined one time but executed many times. This works well with the Tcl byte-code compiler. The disadvantage is that if you modify procedures in these files you have to explicitly source them into the server for these changes to take effect. A built-in URL makes this possible. The `/debug/source` URL accepts a source parameter that indicates what file to load. For safety reasons, it only loads files from the script library directory.

The advantage of putting code into the per-directory "`.tml`" files is that changes are picked up immediately with no effort on your part. The server automatically checks if these files are modified, and sources them each time it processes your templates. However, that code is only run one time, so the byte-code compiler just adds overhead. In general, I try to limit the code in the actual pages to simple procedure calls. Complex code directly in pages cannot be shared, and is more awkward to edit.

Form Handlers

TclHttpd provides alternatives to CGI that are more efficient because they are built right into the server. This eliminates the overhead that comes from running an external program to compute the page. Another advantage is that the Web server can maintain state between client requests in Tcl variables. If you use CGI, you must use some sort of database or file storage to maintain information between requests.

Application Direct Handlers

The server comes with several built-in forms handlers that you can use with little effort. The `/mail/forminfo` URL will package up the query data and mail it to you. You use form fields to set various mail headers, and the rest of the data is packaged up into a Tcl-readable mail message. Example 5 shows a form that uses this handler.

The mail message sent by `/mail/forminfo` is shown in Example 6.

It is easy to write a script that strips the headers, defines a `data` procedure, and uses `eval` to process the message body. Whenever you send data via email, if you format it with Tcl list structure you can process it quite easily.

Template Form Handlers

The drawback of using application-direct URL form handlers is that you have to modify their Tcl implementation to change the resulting page. Another approach is to use templates for the result page that embed a command that handles the form data. The `Mail_FormInfo` procedure, for example, mails form data. It takes no arguments. Instead, it looks in the query data for `sendto` and `subject` values, and if they are present it sends the rest of the data in an email. It returns an HTML comment that flags that mail was sent.

A *self-posting form* is a form that posts the form data to back to the page containing the form. The page embeds a Tcl command to check its own form data. Once the data is correct the page triggers a redirect to the next page in the flow. This is a powerful trick, which I learned from Monty Swiryn of Cuesta Technologies, that you can use to create complex page flows using templates.

Example 5 Mail form results with /mail/forminfo.

```
<form action=/mail/forminfo method=post>
    <input type=hidden name=sendto value=mailreader@my.com>
    <input type=hidden name=subject value="Name and Address">
    <table>
        <tr><td>Name</td><td><input name=name></td></tr>
        <tr><td>Address</td><td><input name=addr1></td></tr>
        <tr><td> </td><td><input name=addr2></td></tr>
        <tr><td>City</td><td><input name=city></td></tr>
        <tr><td>State</td><td><input name=state></td></tr>
        <tr><td>Zip/Postal</td><td><input name=zip></td></tr>
        <tr><td>Country</td><td><input name=country></td></tr>
    </table>
</form>
```

Example 6 Mail message sent by /mail/forminfo

```
To: mailreader@my.com
Subject: Name and Address

data {
    name    {Joe Visitor}
    addr1   {Acme Company}
    addr2   {100 Main Street}
    city    {Mountain View}
    state   California
    zip     12345
    country   USA
```

Of course, you need to save the form data at each step. You can put the data in Tcl variables, use the data to control your application, or store it into a database. TclHttpd comes with a `Session` module that is one way to manage this information.

Example 7 shows the `Form_Simple` procedure that generates a simple self-checking form. Its arguments are a unique id for the form, a description of the form fields, and the URL of the next page in the flow. The field description is a list with three elements for each field: a required flag, a form element name, and a label to display with the form element. You can see this structure in the template shown in Example 8 on page 9. The procedure does two things at once. It computes the HTML form, and it also checks if the required fields are present. It uses some procedures from the `form` module, which is described on page 9, to generate form elements that retain values from the previous page. If all the required fields are present, it discards the HTML, saves the data, and triggers a redirect by calling `Doc_Redirect`.

Example 7 A self-checking form procedure.

```
proc Form_Simple {id fields nextpage} {
    global page
    if {![form::empty formid]} {
        # Incoming form values, check them
        set check 1
    } else {
        # First time through the page
        set check 0
    }
    set html "<!-- Self-posting. Next page is $nextpage -->\n"
    append html "<form action=\"$page(url)\" method=post>\n"
    append html "<input type=hidden name=formid value=$id>\n"
    append html "<table border=1>\n"
    foreach {required key label} $fields {
        append html "<tr><td>"
        if {$check && $required && [form::empty $key]} {
            lappend missing $label
            append html "<font color=red>*</font>"
        }
        append html "</td><td>$label</td>\n"
        append html "<td><input [form::value $key]></td>\n"
        append html "</tr>\n"
    }
    append html "</table>\n"
    if {$check} {
        if {![info exist missing]} {

            # No missing fields, so advance to the next page.
            # In practice, you must save the existing fields
            # at this point before redirecting to the next page.

            Doc_Redirect $nextpage
        } else {
            set msg "<font color=red>Please fill in "
            append msg [join $missing ", "]
            append msg "</font>"
            set html <p>$msg\n$html
        }
    }
    append html "<input type=submit>\n</form>\n"
    return $html
}
```

Example 8 A page with a self-checking form.

```
<html><head>
    <title>Name and Address Form</title>
</head>
<body bgcolor=white text=black>
    <h1>Name and Address</h1>
    Please enter your name and address.
    [myform::simple nameaddr {
        1 name    "Name"
        1 addr1   "Address"
        0 addr2"  "Address"
        1 city    "City"
        0 state   "State"
        1 zip     "Zip Code"
        0 country "Country"
    } nameok.html]
</body></html>
```

Example 8 shows a page template that calls `Form_Simple` with the required field description.

The `form` package

TclHttpd comes with a `form` package that is designed to support self-posting forms. The `Form_Simple` procedure uses `form::empty` to test if particular form values are present in the query data. The `form::value` procedure is useful for constructing form elements on self-posting form pages. It returns:

name="*name*" value="*value*"

The *value* is the value of form element *name* based on incoming query data, or just the empty string if the query value for *name* is undefined. This way the form can post to itself and retain values from the previous version of the page. It is used like this:

<input type=text [form::value *name*]>

The `form::checkvalue` and `form::radiovalue` procedures are similar to `form::value` but designed for checkbuttons and radio buttons. The `form::select` procedure formats a selection list and highlights the selected values. The `form::data` procedure simply returns the value of a given form element.

Experiences with the Server

I have used TclHttpd on two main servers, sunscript.sun.com and www.scriptics.com, and many internal web sites. During a recent week, www.scriptics.com got over 18,000 home page hits, over 200,000 HTTP requests, over 4 gigabytes of data transferred, and over 26,000 page views in the Tcl Resource Center. If you visit http://www.scriptics.com/status you can get a live view of the statistics. This page shows per-minute hit rates over the last hour, per hour hit rates over the last day, and daily hit rates since the server was started. These "hits" are URL requests, which are larger than the number of page views because of images on a page. This traffic is high compared to an average companies site, but low compared to a large portal site.

The current per-minute rates are 100 to 200 hits/minute. Previously it was as high as 500 hits/minute due to the large number of images on our pages. We recently split the image traffic to another web server. On two occasions a bug in the server trapped a remote client by accidentally redirecting it to the same page that the client was requesting. This caused the client to fetch the same page again and again. When this happened, I observed sustained per-minute hit rates of over 700 hits/minute. Both problems were fixed by loading an explicit redirect that aimed the client at the page they really wanted; it did not require a server restart.

I performed some basic comparisons of servers on a test network of four machines on 100Mbit ethernet. The machines were a dual-processor Sparc-20 running at 75 MHz, an Ultra-5 Sparc running at 270 Mhz, a Pentium II running Linux at 400 MHz, and a Pentium III running Windows NT at 450 MHz. TclHttpd was run on all platforms, while Apache, Netscape, AOLserver, and IIS were run on a subset of the platforms. No performance tuning was done on any of the servers.

The test simply performs a number of HTTP requests to various URLs: a URL implemented by a simple Tcl procedure, a small image, a medium sized image, and a large image.

Three charts are shown in the appendix. The Dell-450 chart shows the performance of TclHttpd and IIS on the fastest machine. TclHttpd adds overhead that is relatively high for small transfers (about 3.5 msec vs 12.5 msec for 200 bytes) and less so for big transfers (about 23 msec vs 37 for 120K.) The Dynamic Pages chart shows the performance of dynamic pages. This shows the obvious benefit of building page generation right into the server. The fastest is AOLserver at about 8 msec. The mod_tcl plugin for Apache was close behind at 11 msec. TclHttpd was about 23 msec, and CGI from Apache was about 72 msec. The 32 Kbyte chart compares the time for all different servers to deliver a 32Kbyte image. TclHttpd runs from 2 to 3 times slower than the fastest server on the platform for this sized transfer.

Perhaps the most notable experience from using TclHttpd on www.scriptics.com is that it is extremely robust. The server is a Sparc-20 running Solaris 2.5.1, and in a two year period I experienced one OS crash, and two or three occasions where I was forced to reboot the machine for various administrative reasons. Tcl, of course, never crashed, so TclHttpd ran for months at a time.

The other exciting thing about TclHttpd is the ability to modify the application without restarting the server. In the early days at sunscript.sun.com I fixed various bugs in the core TclHttpd code. It is more common that the bug fixes are in various the form handlers we have at www.scriptics.com. When a page generates a Tcl error, an error page is displayed in the browser. This contains a form with two options: view the `errorInfo` from the error, or mail that information to `webmaster@scriptics.com`. Of course, TclHttpd continues to function. We can usually diagnose the problem quickly just by looking at `errorInfo`. For difficult bugs we start another copy of the server and connect to it remotely with TclPro Debugger. Once the bug is fixed we simply load new code into the server to fix the problem. One danger of continuously modifying the server is that you can have a server that is running fine but cannot be restarted because of bugs in the startup code. After significant server changes I either restart the server or test the startup sequence by starting the application on a different port.

There have been many applications of TclHttpd as an embedded server. We use it for the Scriptics License Server that implements our shared licences for TclPro.

Current work on TclHttpd includes exploiting the threading capabilities of Tcl 8.2. A threaded server can eliminate the need to use CGI for long-running template code. In addition, Matt Newman has used the built-in stacked channel support in Tcl 8.2 to create a clean SSL extension to the server.

Related Work

There are a number of other interesting Tcl-based Web servers. Karl Lehenbaur presented a paper on the NeoWebScript Apache plugin in an earlier Tcl/Tk conference. David Welton wrote the mod_dtcl plugin for Apache. Probably the most mature Tcl-based web server is the AOLserver. This has used a multi-threaded version of Tcl for some time: first 7.4, then 7.6, and now 8.2. All of these support HTML+Tcl templates, although the syntax used to embed Tcl on the page varies somewhat from server to server.

What makes TclHttpd novel is the ability to embed the server into another application. The event-based model simplifies the integration of the server into the application. In contrast, an Apache or IIS plugin is forced to deal with the multi-process or multi-thread architecture of the hosting web server. Once you have embedded TclHttpd, you have a variety of ways to integrate it with your application, including Application-Direct URLs, custom domain handlers, document handlers, and dynamic page templates.

Web Links

The TclHttpd home page:
`http://www.scriptics.com/products/tclhttpd/`

The AOLserver home page:
`http://www.aolserver.com/`

The mod_dtcl home page:
`http://comanche.com.dtu.dk/dave/`

The NeoWebScript home page:
`http://www.NeoSoft.com/neowebscript/`

Appendix A: Templates for Site Structure

This appendix shows a simple template system used to maintain a common look at feel across the pages of a site. Example 9 shows a simple one-level site definition that is kept in the root `.tml` file. This structure lists the title and URL of each page in the site:

Each page includes two commands, `SitePage` and `SiteFooter` that generate HTML for the navigational part of the page. Between these commands is regular HTML for the page content. Example 10 shows a sample template file:

The `SitePage` procedure takes the page title as an argument. It generates HTML to implement a standard navigational structure. Example 11 has a simple implementation of `SitePage`:

The `foreach` loop that computes the simple menu of links turns out to be useful in many places. Example 12 splits out the loop and uses it in the `SitePage` and `SiteFooter` procedures. This version of the templates creates a left column for the navigation and a right column for the page content:

Of course, a real site will have more elaborate graphics and probably a two-level, three-level, or more complex tree structure that describes its structure.You can also define a family of templates so that each page doesn't have to fit the same mold. Once you start using templates, it is fairly easy to change both the template implementation and to move pages around among different sections of your web site.

There are many other applications for "macros" that make repetitive HTML coding chores easy. Take, for example, the link to `/ordering.html` in Example 10. The proper label for this is already defined in `$site(pages)`, so we could introduce a `SiteLink` procedure that uses this:

If your pages embed calls to `SiteLink`, then you can change the URL associated with the page name by changing the value of `site(pages)`. If this is stored in the top-level `".tml"` file, the templates will automatically track the changes.

Example 9 A one-level site structure.

```
set site(pages) {
    Home                   /index.html
    "Ordering Computers"/ordering.html
    "New Machine Setup" /setup.html
    "Adding a New User" /newuser.html
    "Network Addresses" /network.html
}
```

Example 10 A HTML + Tcl template file.

```
[SitePage "New Machine Setup"]
This page describes the steps to take when setting up a new
computer in our environment. See
<a href=/ordering.html>Ordering Computers</a>
for instructions on ordering machines.
<ol>
<li>Unpack and setup the machine.
<li>Use the Network control panel to set the IP address
and hostname.
<!-- Several steps omitted -->
<li>Reboot for the last time.
</ol>
[SiteFooter]
```

Example 11 SitePage template procedure. Simple horizontal menu along the top of the page.

```
proc SitePage {title} {
    global site
    set html "<html><head><title>$title</title></head>\n"
    append html "<body bgcolor=white text=black>\n"
    append html "<h1>$title</h1>\n"
    set sep ""
    foreach {label url} $site(pages) {
        append html $sep
        if {[string compare $label $title] == 0} {
            append html "$label"
        } else {
            append html "<a href='$url'>$label</a>"
        }
        set sep " | "
    }
    return $html
}
```

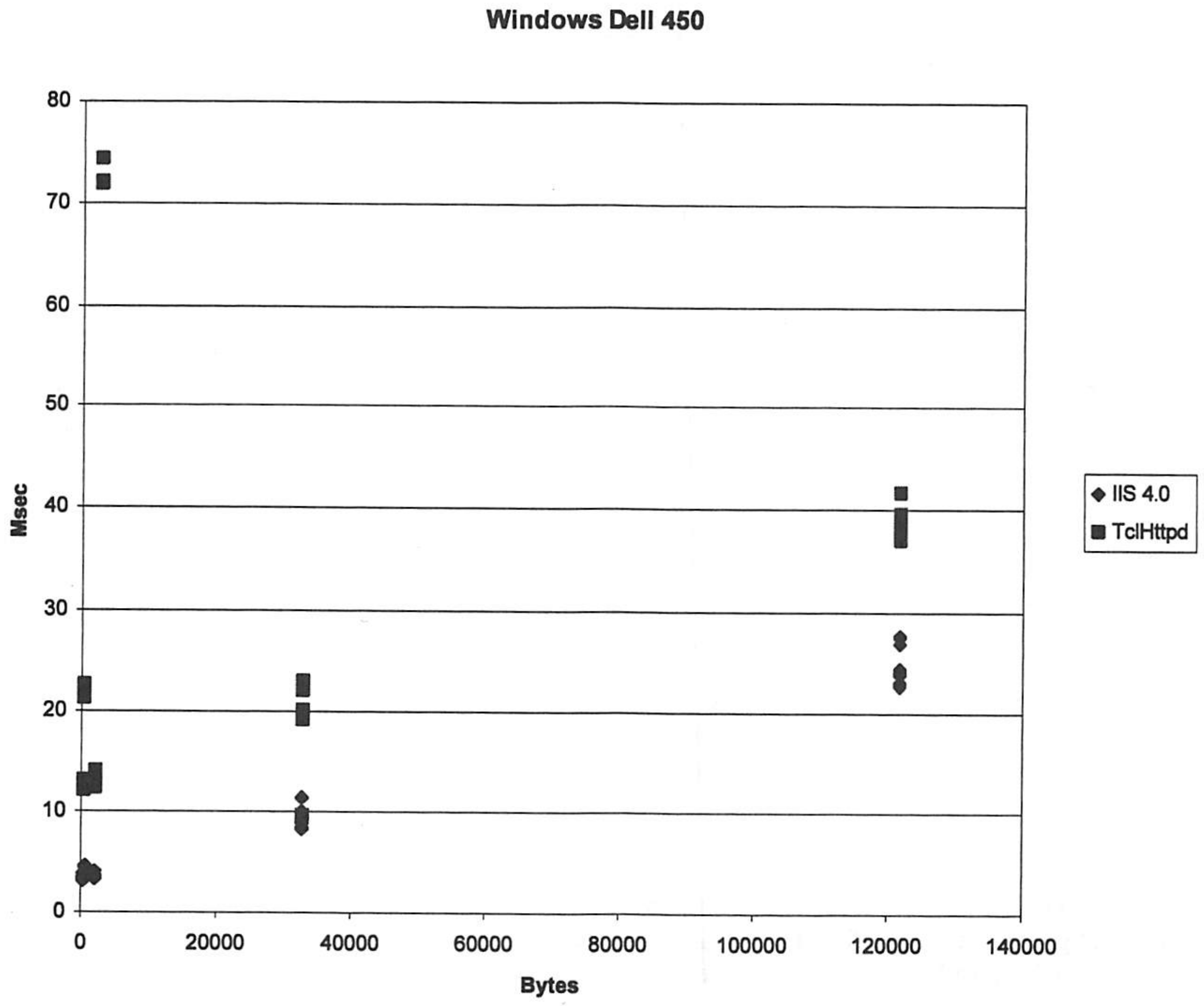
Example 12 SiteMenu and SiteFooter template procedures. Two-column format with menu in the left column.

```
proc SitePage {title} {
    global site
    set html "<html><head><title>$title</title></head>\n\
        <body bgcolor=$site(bg) text=$site(fg)>\n\
        <!-- Two Column Layout -->\n\
        <table cellpadding=0>\n\
        <tr><td>\n\
        <!-- Left Column -->\n\
        <img src='$site(mainlogo)'>\n\
        <font size=+1>\n\
        [SiteMenu <br> $site(pages)]\n\
        </font></td><td>\n\
        <!-- Right Column -->\n\
        <h1>$title</h1>\n\
        <p>\n"
    return $html
}
proc SiteFooter {} {
    global site
    set html "<p><hr>\n\
        <font size=-1>[SiteMenu | $site(pages)]</font>\n\
        </td></tr></table>\n"
    return $html
}
proc SiteMenu {sep list} {
    global page
    set s "" ; set html ""
    foreach {label url} $list {
        if {[string compare $page(url) $url] == 0} {
            append html $s$label
        } else {
            append html "$s<a href='$url'>$label</a>"
        }
        set s $sep
    }
    return $html
}
```

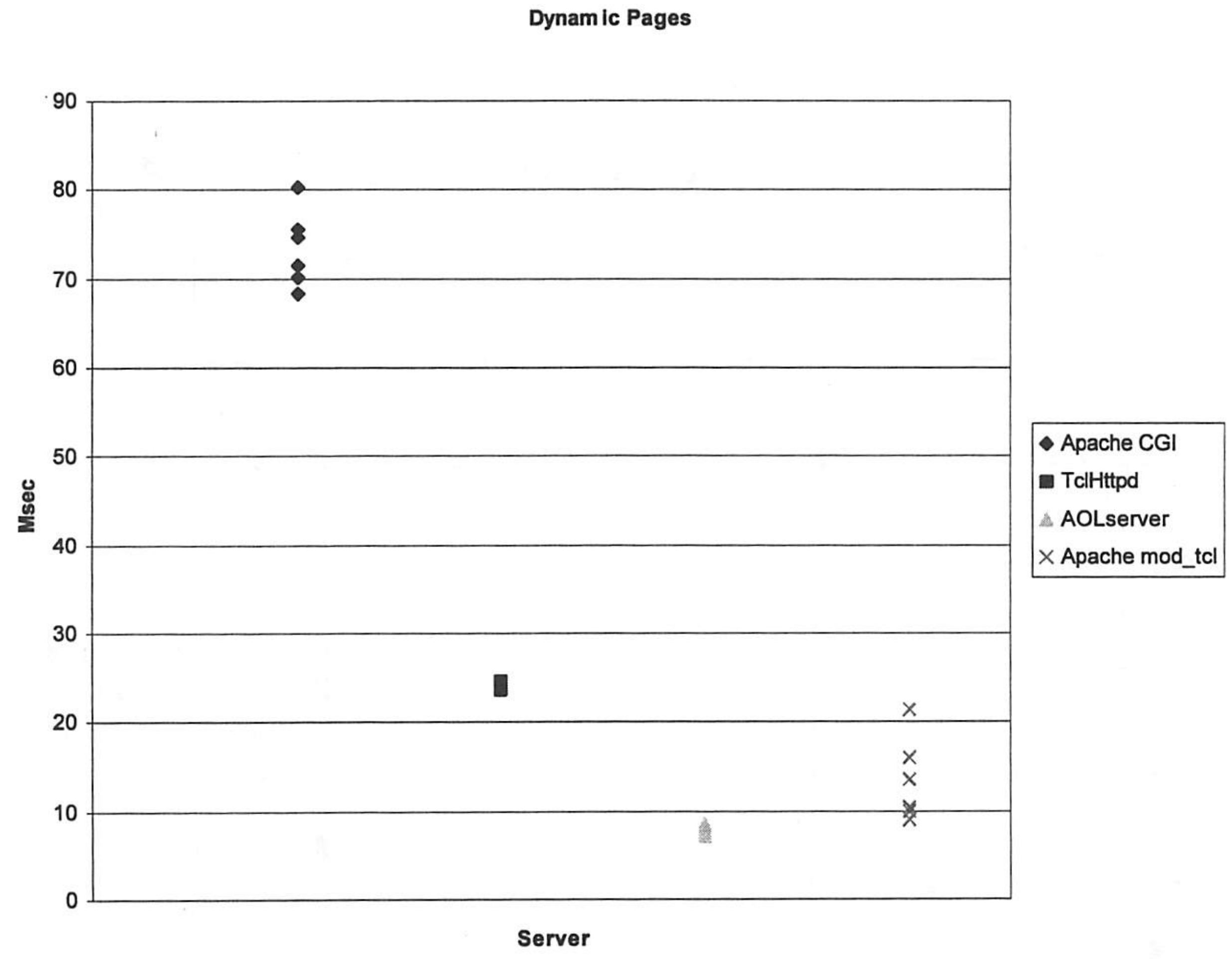
Example 13 The SiteLink procedure.

```
proc SiteLink {label} {
    global site
    array set map $site(pages)
    if {[info exist map($label)]} {
        return "<a href='$map($label)'>$label</a>"
    } else {
        return $label
    }
}
```

Appendix B: Performance Charts



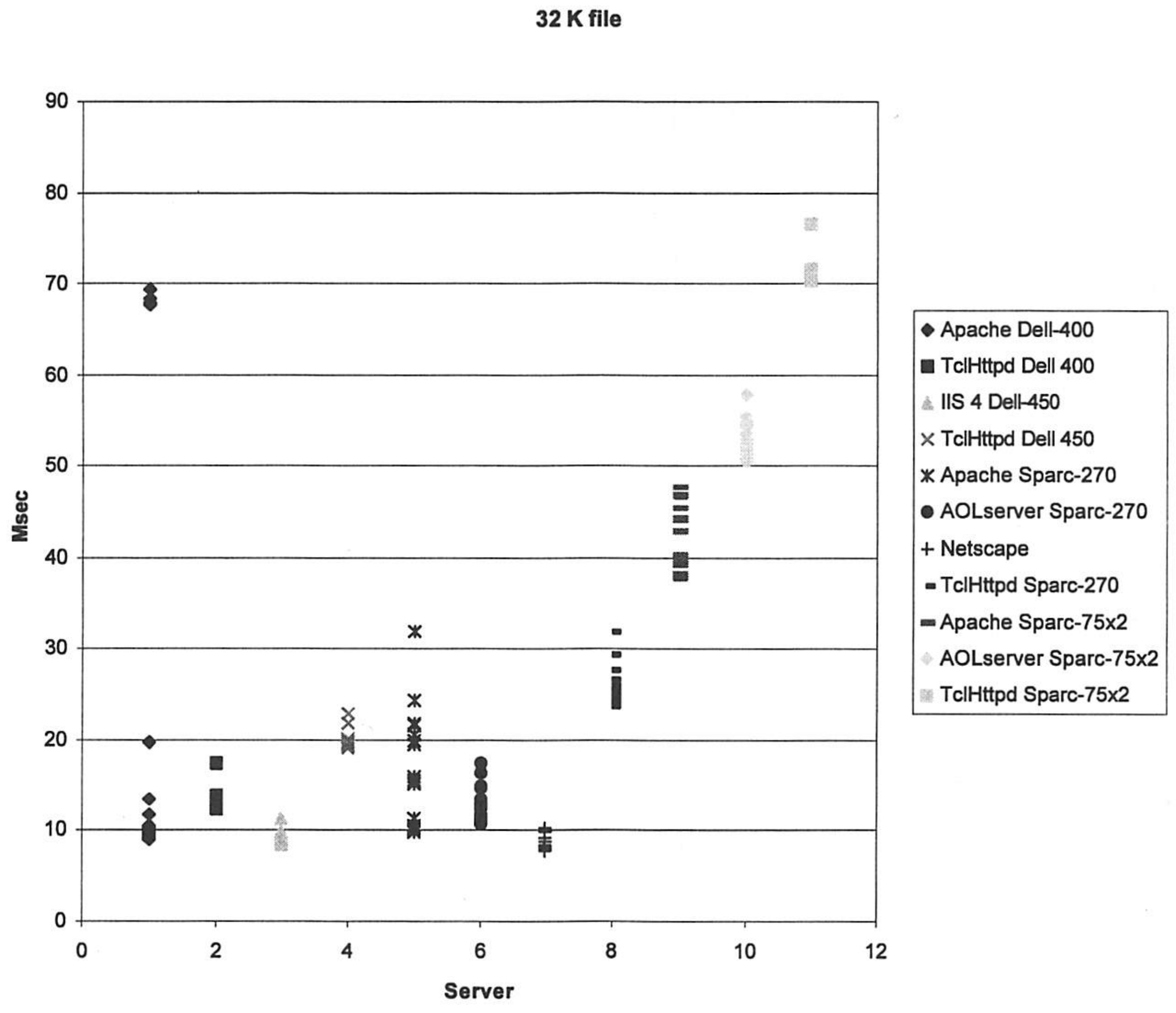
This chart shows two web servers, IIS 4 and TclHttpd, running on a 450 MHz Pentium-III under Windows NT. There is a plot for each run so you can see the variation across runs. Each run fetched the same URL repeatedly from the server using a single-threaded client. The runs performed either 100, 200, or 1000 repetitions, although these are not distinguished in the graph. These runs include some dynamic pages as well as static. The outlying points for TclHttpd are CGI scripts. There were no CGI tests done on IIS.



This chart shows the cost of creating dynamic pages on different platforms. The tests were repeated several times, and points are plotted for each run. All tests were run on the Sparc-270. CGI is slowest, of course, because Apache must fork a process. AOLserver is fastest, with the mod_tcl plugin for Apache close behind. TclHttpd is about three times faster than Apache CGI, and AOLserver is about 10 times faster than Apache CGI.

The dynamic page was very trivial, equivalent to:

puts “hello, world”



This figure compares all web servers when fetching a 32 Kbyte image file. Note that both the hardware and the web server are changing. For each hardware platform, TclHttpd and one or more other servers were compared. Overall TclHttpd runs from 1.5 to 3 times slower for moderate sized transfers.

TkGecko: A Frill-Necked Lizard

Steve Ball
Zveno Pty Ltd
Eolas Technologies, Inc
Steve.Ball@zveno.com

Abstract

The Mozilla Open Source project has made a full-featured, fast Web browser, Netscape's Navigator, available in source form to the Internet community. A major component of the project, and an early released package, is the NewLayout (a.k.a. Gecko) HTML/XML rendering engine. One feature of this module is that it is designed to be embeddable in any application. This characteristic is quite compatible with Tcl/Tk.
TkGecko is a project to create a Tk extension that allows the NewLayout rendering engine to be embedded in a Tk application as a Tk widget. Ideally, this new widget will be as completely configurable as all other Tk widgets, with all configuration options able to be queried and changed at run-time. The widget features methods and configuration options that allow the functions of the NewLayout rendering engine to be accessed and changed from a Tcl script.
Currently the TkGecko project is still in its infancy. This paper aims to define the goals of the project and expected milestones.

Keywords

Netscape Navigator, Mozilla, NewLayout, NGLayout, Gecko, Tcl, Tk, TEA.

1. Introduction

In 1997 Netscape Communications Corporation [1] released the source code for their Netscape Navigator Web browser to the Internet community, under the Netscape/Mozilla Public License (NPL or MPL) [2], a license that allows free use of the source, redistribution and modification. Netscape has setup an independent organisation to develop the next release of Netscape Navigator under the Open Source banner called Mozilla.org [3]. Currently, the majority of developers working on Mozilla are employed by Netscape (approximately 120 engineers), but there are some employed by the other organisations (about 25 engineers).

The first major "product" release from Mozilla.org is Gecko, the core HTML/XML page rendering engine for the browser. Gecko is known more formerly as the NewLayout module. This module is written in C++ (in fact, all of Mozilla is written in C++, but there does exist a separate project to reimplement Mozilla in Java), and is designed to be embeddable inside other applications, for example as a HTML-based help viewing system. There are a few examples of Gecko being embedded in applications, such as an ActiveX component [4] (which entirely replaces the Internet Explorer control!) and the DocZilla viewer [5].

Eolas Technologies, Inc. [6] are sponsoring work to create a Tk extension which embeds Gecko as a Tk widget. The working title for this project is "TkGecko". The extension defines a new Tk widget class called "newlayout". The availability of this extension will provide a high-quality viewer of HTML and XML documents to be used by Tk applications, with full support for all of the current Web standards, such as CSS, JavaScript, DOM and XSL. While most of the major Web standards will be supported, it is not clear at this stage whether the extension will be able to handle Java applets or browser plugins. The former will require a Java Runtime Engine (JRE) to be included in the extension.

Although TkGecko will allow the creation of a general-purpose Web browser, that is not the primary motivation for this project since at least two general-purpose Web browsers already exist. Instead, the aim is to support application developers wishing to incorporate a HTML/XML viewer into their products, often for the purpose of displaying help documents. It is certainly now the case that Web browsers are now so ubiquitous that an application developer can assume their availability and simply launch a browser to display a document. However, many developers wish to have a tighter integration of the display of help documents, particularly for context-sensitive help, with their application. An embedded viewer is necessary to satisfy this requirement.

This approach to displaying documents has been already been anticipated by Microsoft, who have made their Internet Explorer browser available not just as a stand-alone browser but also as an ActiveX control [13]. Hence applications running on a Microsoft Windows platform can easily embed the Internet Explorer ActiveX control to realise an embedded HTML/XML document viewer. Alternatively, there is an equivalent ActiveX control for Mozilla [4]. However, this is not a cross-platform solution and therefore the need to have an easy method of embedding the Gecko rendering engine.

1.1. Distribution

In the first phase of the project, the basic embedding of the NewLayout widget as a Tk widget will be achieved. This phase includes writing all of the necessary configuration files to link the numerous required libraries into a dynamically loadable shared library, for the Linux (i386) and Microsoft Windows 95/98/NT/2000 platforms. The new Tk widget class is called `newlayout`.
Subsequent phases of the project will implement configuration options and methods for the newlayout widget, as detailed in the section "Future Directions". It is also an aim of the project to extend the package to the MacOS and LinuxPPC platforms and architectures.
Packaging TkGecko for distribution to interested parties presents some problems. The source code alone for Mozilla is approximately 19MB, but to actually build the browser requires almost 1GB of disk space. However, these requirements may be reduced by customising the build process to compile only what is needed for the Gecko engine. Customising the Mozilla build process has not yet been attempted. Alternatively, the binaries of the Gecko engine are only approximately 1.4MB in size so binary distributions for the platforms specified above are the currently favoured form for making the extension available.

1.2 Terminology

There are numerous names and terms surrounding the Mozilla project and its components. The Gecko HTML/XML rendering engine is more correctly known as the NewLayout or NGLayout module. Netscape Communications Corp. marketing droids originally dubbed it "Gecko", but the developers themselves don't particularly use that term.

2. Mozilla Architecture

In order to design and implement a Tk extension to embed Mozilla, it is vitally important to understand the architecture of the Mozilla browser. While each of the technologies required to construct a Web browser are relatively simple, combining them together into a sophisticated Web browser makes Mozilla a complex piece of software engineering.
Of particular interest in the Mozilla architecture is the NGLayout module. The overall goal of the module is to provide high-quality rendering of HTML and XML documents, while at the same time achieving high-performance in page display. Documents themselves have a complex structure, especially when HTML frames (subdocuments) and tables are taken into account. This module has been designed to allow for the complexities of page display, as well as supporting the JavaScript page scripting language and dynamically loaded content viewers, for rendering different content types such as image formats. It is beyond the scope of this paper to undertake a detailed study of the module, nor to explain how high performance is achieved.
Following is a brief overview of the modules which are of the most importantance to the NewLayout extension. Figure 1 shows their relationship.

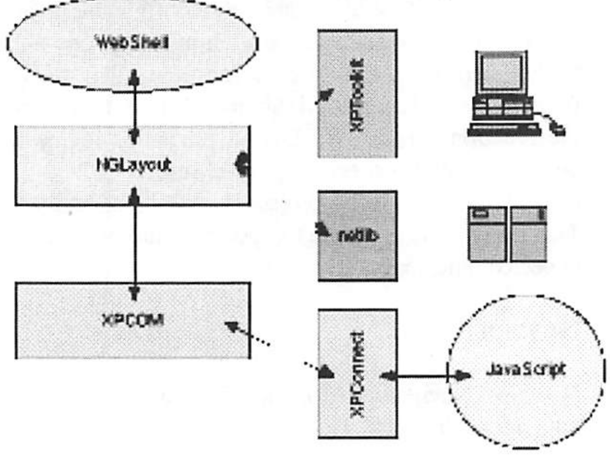


Figure 1: Mozilla Architecture

Underpinning the entire Mozilla architecture is the XPCOM (Cross-Platform Component Object Module) module. XPCOM is the same in concept to Microsoft's COM, but is much simplified and implemented across all operating system platforms. In fact, XPCOM aims to be binary-compatible with COM. One feature of XPCOM is that because it binds component interfaces together at runtime it allows parts of Mozilla to be used separately and to replace an implementation of a component with another seamlessly. This characteristic is very useful for building and shipping the TkGecko extension.

Another module of interest is XPConnect, which provides a bridge between XPCOM and JavaScript. This module allows JavaScript scripts to access and manipulate XPCOM objects, as well as allowing a JavaScript program to provide the implementation for a XPCOM object.

In order to be able to write portable User Interface code, the Mozilla project is developing the XPToolkit (Cross-Platform Toolkit) module. This module uses XUL, the XML User Interface Language, to describe user interfaces. As the name suggests, a XUL document provides the description of the desired widgets and their layouts as a XML document. XPToolkit parses the XML document and then assembles and displays the appropriate native widgets for the interface. The implication of this approach is that building a user interface becomes the simple task of writing a document.

At the lower level of windowing and drawing primitives, the Cross-Platform Front-End (XPFE) module provides a portable interface to the underlying windowing system of the machine platform. On Windows and Macintosh this is obviously the operating system's native windowing facilities, but on Unix/X Window the situation is more complicated due to the plethora of available toolkits. A number of toolkits are supported, but the primary toolkit used by the Mozilla browser is GTk+. However, the embedding interface also supports the direct use of Xlib. An initial approach to embedding the NewLayout in Tk used the GTk+ embedding interface, but since Tk also uses Xlib directly, the TkGecko extension now instructs Mozilla to use the Xlib toolkit. This is discussed in more detail below. It was recently announced that the WebShell interface is to be redesigned, and that the alternative toolkits may be dropped in favour of concentrating on the GTk+ toolkit. USENET newsgroup discussion of the new design also supports retaining the Xlib interface. As a result the TkGecko project will need to reassess its approach to embedding the NewLayout module once the module redesign is underway.

Networking functions and protocol support are provided by the netlib module. There is also a new project to improve netlib called "Necko". These modules handle the transfer of document data, and use threads to deal with latencies. As of Milestone 9 Necko is now used with Mozilla. Integrating the netlib/Necko event model with the Tcl Notifier may be an issue for the TkGecko project, but at this early stage has proven to be simple and straight-forward.

2.1 NewLayout Module

On top of all of these modules (and some others not mentioned for the sake of brevity) is the NewLayout module, the subject of the TkGecko project. The aim of this module is to provide a small, fast rendering engine for Web documents.

The most important layer of the NewLayout module is the WebShell interface. This module also provides the embedding function of the NewLayout module. A WebShell is used to display each subdocument (frame) in a document. WebShells may be nested, and the initial WebShell is known as the root WebShell. Since WebShells may contain other WebShells, the nsIWebShell class is subclassed from the nsIWebShellContainer class. The nsIWebShellContainer class provides additional methods that provide notification of events in the contained WebShell(s), such as the start of loading documents, the end of a document load, and so on.

At times the NewLayout module may need to query the parent window of the root WebShell for a document. This is done using the `nsIBrowserWindow` interface. For example, NewLayout may wish to set the title of the window, or update a progress bar. To support these functions, TkGecko will also need to implement an interface to the `nsIBrowserWindow` class. This is not yet currently done.

The WebShell interface provides a number of methods for manipulating the (sub-)document. A few of these methods include:

`LoadURL`

: Load a document, given a URL

`Stop`

: Terminate loading the document

`Back`

: Load the previous document on the history list

`Forward`

: Load the next document on the history list

`GetChildCount`

: Return the number of WebShells which are children of the current WebShell

`GetParent`

: Return tne parent WebShell of the current WebShell

AddChild

Add a new WebShell

3. `newlayout` Tk Widget

The newlayout Tk widget is the main feature of the TkGecko extension. Using this widget, a Tk application developer is able to easily embed a Mozilla WebShell widget in a Tk user interface.

3.1 TEA Compliance

A major objective of the TkGecko extension is to make embedding the Mozilla NewLayout module as easy as possible for Tk application developers. The first step to achieving this goal is to make the package TEA (Tcl Extension Architecture) compliant [7]. TEA compliant packages are able to be dynamically loaded into any version of a Tcl interpreter in a forward-compatible fashion without re-linking. Given that compiling Mozilla is a non-trivial undertaking this feature is very useful.

TEA compliance proved to be very easy to do. Modifying the autoconf and automake configuration scripts from the TEA sample extension provided by Scriptics was straight-forward. Mozilla also makes heavy use of autoconf and GNU Make, but in a fashion not entirely compatible with the manner in which TEA does. For example, Mozilla uses Makefiles in a hierarchical configuration, whereas TEA uses automake. Integrating the automated Mozilla configuration and build system with TEA's proved to be a difficult problem, which has not yet been completed to a satisfactory conclusion.

3.2 `newlayout` Widget Architecture

TkGecko has two modules, the main module, written in C, is `newlayout.c` which provides the interface with the Tcl interpreter and Tk. Since Mozilla is written in C++ it is necessary for the TkGecko extension to include another module to provide the C++ interface. This is the `tkmozilla.cpp` module. The TkGecko extension architecture is shown in figure 2.

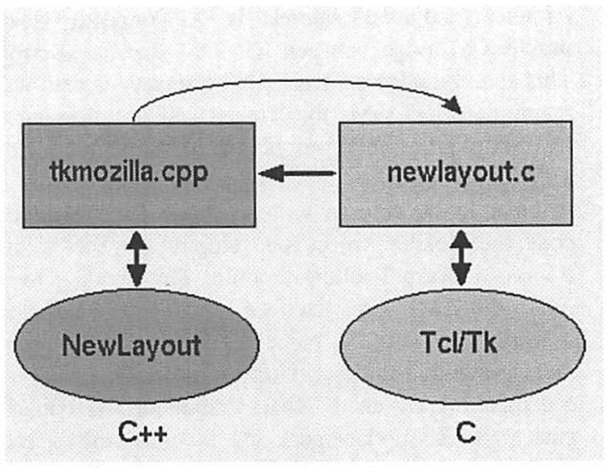


Figure 2: TkGecko Module Structure

The `newlayout.c` module takes care of all of the Tcl/Tk housekeeping. This includes package and stub initialisation, widget class creation command, widget command, widget method implementations, configuration option processing, Tk event handlers and so on. When interaction with the Mozilla NewLayout module is required, a wrapper function in the `tkmozilla.cpp` module is called. The module also includes a number of "call-in" procedures which the `tkmozilla.cpp` may invoke in response to certain events occurring. These procedures check whether a callback has been defined for the event and evaluate the callback if present, otherwise the procedure takes no action.

The `tkmozilla.cpp` module provides an interface to the Mozilla NewLayout module. Each procedure in this module provides an interface that is callable from C code, ie. the `newlayout.c` module. This approach is used to expose the functionality of the Mozilla NewLayout module to the Tcl/Tk interpreter. In addition, this module will "register" procedures to be called to handle certain events. In some cases these are registered explicitly as callbacks and in other cases they become associated with events by subclassing certain NewLayout classes. In either case the events are handled by explicitly invoking a call-in procedure in the `newlayout.c` module.

3.3 Embedding The WebShell

Embedding a WebShell requires only a few steps to be taken [12]:

1. Register the NGLayout libraries
2. Create an event queue
3. Initialize the network library
4. Create a WebShell

The first three functions are performed by the `newlayout.c` module upon loading of the package by

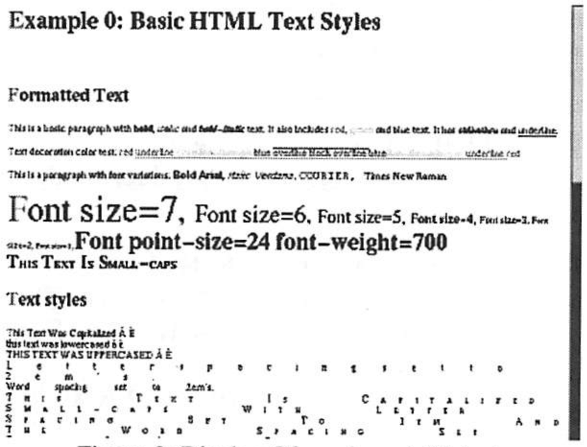
calling `tk_mozilla_init`, a procedure provided by the `tkmozilla.cpp` module. On Unix Necko functions using a `select` style interface. The file descriptor used by Necko is retrieved and added to the Tcl Notifier as an event source, with a callback to the `newlayout.c` module.
The actual creation of a WebShell widget is performed each time a newlayout widget is instantiated. Tk widgets may have several instantiations, so the TkGecko extension must support the creation of multiple WebShell widgets. The WebShell creation interface requires the Xlib window identifier, so the actual creation of the WebShell widget must be delayed until the newlayout Tk widget is first mapped to the screen. This has the unfortunate effect of causing a time lag between the appearance of the widget on screen and in the initialisation of the WebShell widget. This effect seems to be further complicated by the threading involved.

4. Using TkGecko

Using TkGecko in a Tk application is now just as simple as any other Tk widget. Consider the following script examples.

```
package require newlayout

newlayout .mozilla
grid .mozilla

.mozilla configure -url
file:///home/steve/samples/test0.html
```

Figure 3 shows the user interface generated by this script. Note carefully the plain scrollbar present on the right-hand side. This scrollbar is added by the NewLayout module and currently TkGecko is not able to disable it or control the scrolling.



Example 0: Basic HTML Text Styles

Formatted Text

This is a basic paragraph with bold, italic and bold-italic text. It also includes red, green and blue text. It has strikethru and underline.

Text decoration color test: red underline blue overline black overline blue underline red

This is a paragraph with font variations. Bold Arial, Italic Verdana, COURIER, Times New Roman

Font size=7, Font size=6, Font size=5, Font size=4, Font size=3, Font size=2, Font size=1, Font point-size=24 font-weight=700
This Text Is Small-caps

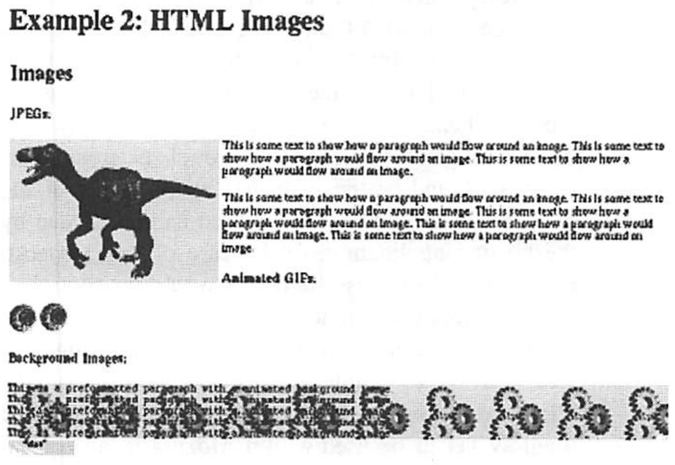
Text styles

This Text Was Capitalized Å È
this text was lowercased å è
THIS TEXT WAS UPPERCASED Å È
L e t t e r s p a c i n g s e t t o 2 e m ' s .
Word spacing set to 2em's.
T h i s T e x t I s C a p i t a l i z e d S m a l l - c a p s W i t h L e t t e r S p a c i n g S e t T o 1 e m A n d T h e W o r d S p a c i n g S e t

Figure 3: Display Of newlayout Widget

```
package require newlayout

newlayout .mozilla
grid .mozilla

.mozilla configure -url
file:///home/steve/samples/test2.html
```

Figure 4 shows the user interface generated by this script. Due to a malfunction in the NewLayout timer library the animated GIF images do not function correctly (anyone who has seen the bloodshot eyes in action would consider this to be an improvement).



Example 2: HTML Images

Images

JPEGs.

This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image.

This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image. This is some text to show how a paragraph would flow around an image.

Animated GIFs.

Background Images:

This is a preformatted paragraph with animated background image

Figure 4: Display Of newlayout Widget

4.1 Current Status

At this early stage of the project, loading the extension and displaying a document is the only feature of the TkGecko extension that works. Loading new URLs also works.
The Tk and Mozilla event systems have not been hooked up to the stage where events can flow freely, thus hyperlinks do not work, nor does widget redraw or resize. Instantiating multiple newlayout widgets crashes the application. These bugs and missing features are the highest priority for the project, in order to achieve basic functionality of the widget.

5. Development Plan

Further work is scheduled for TkGecko, with the aim of making the extension generally usable and stable by the end of 1999. The development schedule includes the following milestones:

- Fix crashing bugs and malfunctioning timer libraries.
- Add event processing.
- Fix configuration options.

- Add callbacks for WebShell widget events: BeginLoadURL, EndLoadURL, Progress, and so on.
- Add methods to query and manipulate document content and structure.
- Add an event binding mechanism for document content similar to tags in Tk Canvas and Text widgets.

6. Future Directions

Beyond and apart from the planned development of this project, there are some further areas of research to investigate. Some of these are detailed below. The project will also track developments in Mozilla itself.
As mentioned above, the WebShell interface is to be redesigned and this should lead to more functionality of the widget being exposed through external interfaces. It should be the case that every aspect of the widget's function can be monitored and overridden by the hosting application. In the case of the TkGecko extension, all of these functions would be controlled by Tcl scripts via callbacks.
Firstly, as mentioned above, another interesting project will be to implement a Tcl version of the XPConnect module. The goal of this work would be to allow Tcl to be used within Mozilla in an equivalent way to JavaScript.
Secondly, XUL may have some significant ramifications for Tk. It should be possible to implement XUL tools as Tk applications, such as an interface designer. Another possibly useful tools would be to dump a Tk interface as a XUL document, and be able to recreate the interface from the document in Tk. This would have the advantage to the designer of being able to interactively prototype the interface using Tk first.

6. Related Work

Gecko supports the display of XML documents as well as HTML documents, and so has a number of parsers and modules to support this function. These include James Clark's expat XML parser, a Tcl interface to which has already been created [8].
Scripting XML documents within the browser is achieved using the DOM [9], and Mozilla has an implementation of the DOM in C++ for use by JavaScript. A possible future project may be to create a separate Tcl extension to provide an interface to this DOM implementation. The extension would implement the TclDOM API [10], and so maintain compatibility with existing TclDOM scripts. This would be an alternative to tDOM [11]. The content of a WebShell widget is itself a DOM object, so such an interface may be used to manipulate the content of a widget using a Tcl script.

7. Conclusion

Netscape Communications Corporation have provided an opportunity to create a new Tk widget by releasing the source code of Netscape Navigator under an Open Source license. This Open Source project is commonly known as Mozilla.
The TkGecko project has created a Tk extension which creates a new Tk widget, newlayout, to embed the core rendering engine of the Mozilla browser, NGLayout, in a Tk window.
The newlayout widget has minimal functionality at present, but will eventually expose all of the interfaces of the Mozilla NewLayout module to Tk application scripts. Further, the project may also make Mozilla itself scriptable using Tcl/Tk in addition to JavaScript.

8. References

[1] Netscape Communications Corporation. http://www.netscape.com/

[2] Netscape Public License, Mozilla Public License. http://www.mozilla.org/NPL/

[3] Mozilla.org. http://www.mozilla.org/

[4] Mozilla ActiveX Component. Adam Lock. http://www.iol.ie/~locka/mozilla/mozilla.htm

[5] DocZilla Viewer. CiTEC. http://www.doczilla.com/

[6] Eolas Technologies, Inc. http://www.eolas.com/

[7] Tcl Extension Architecture. http://www.scriptics.com/products/tcltk/tea/

[8] *XML Support For Tcl.* S. Ball. Proceedings of the Sixth Annual Tcl/Tk Conference. September 14-18 1998, San Diego CA USA. http://www.zveno.com/zm.cgi/in-tclxml/

[9] Document Object Model. L. Wood, et al.
World Wide Web Consortium Recommendation.
http://www.w3.org/DOM/

[10] TclDOM. S. Ball.
http://www.zveno.com/zm.cgi/in-tclxml/in-tcldom/

[11] *tDOM - a XML/DOM/XPath implementation for Tcl.* J. Loewer.
http://sdf.lonestar.org/~loewerj/tdom.cgi

[12] *Extending Mozilla Or How To Do The Impossible.* J. Stenback, H. Toivonen.
http://www.doczilla.com/development/extmoz.html

[13] *WebBrowser Control.* Microsoft Corp.
http://msdn.microsoft.com/workshop/browser/webbrowser/wbentry.asp

Scriptics Connect

Eric Melski
Scriptics Corporation
ericm@scriptics.com
Scott Stanton
Scriptics Corporation
stanton@scriptics.com
John Ousterhout
Scriptics Corporation
ouster@scriptics.com

Abstract

Scriptics Connect is a commercial product from Scriptics Corporation that provides an XML-based platform for business-to-business applications. It takes advantage of XML as a standard mechanism for formatting structured data, and uses standard World Wide Web servers and Tcl to provide the infrastructure needed to support business-to-business applications. In addition, it uses Tcl to greatly simplify the task of writing XML-based business-to-business applications.

1 Introduction

XML, the eXtensible Markup Language, has the potential to revolutionize the way that businesses communicate and do business with each other. However, XML alone lacks the infrastructure needed to make it successful. In addition, because of the limitations of the tools available, it is difficult to write XML-based busines-to-business applications. Scriptics Connect was designed to address these issues. Our primary goals when developing Scriptics Connect were:

- Provide the missing infrastructure needed to enable XML-based business-to-business applications. Specifically, provide a means of transporting XML documents and a means of integrating XML-based applications with existing applications.

- Reduce the complexity and difficulty of creating XML-based business-to-business applications

In order to meet these goals, we made use of standard World Wide Web servers to provide a transport mechanism, and we made use of Tcl to provide integration facilities and to simplify XML-based application programming. We chose Tcl as the language of implementation for several reasons. First, Tcl is a natural fit to XML, because it has strong string processing capabilities. Second, the wide variety of extensions available for Tcl make it a good fit for business-to-business applications, which all require some level of integration with existing resources. Finally, Tcl enabled us to develop our application far more rapidly than would have been possible with many other languages.

In this paper, we will begin with a description of XML and why it is interesting for business-to-business applications. Then we will give an overview of Scriptics Connect and the particular problems we wanted to address with the system. We will describe in detail the programming abstractions that we implemented to make XML programming and resource integration easy. Finally, we will discuss problems with our implementation, the lessons we learned from implementing the system, and our future plans for Scriptics Connect.

2 XML

XML, the eXtensible Markup Language, is similar to HTML, the HyperText Markup Language of World Wide Web fame. Both are markup languages; that is, they are used to mark sections of a document with semantic and stylistic information, to allow the document to be better rendered or understood by a machine. HTML, however, is limited because it restricts the types of markup that can be used. It

```
<HTML>
<BODY>
<P>
Ship to:<BR>
XYZ Corporation<BR>
2867 Coast Avenue<BR>
Mountain View, CA 94043<BR>
<TABLE>
    <TR>
        <TH>Artist</TH>
        <TH>CD</TH>
    </TR>
    <TR>
        <TD>Weezer</TD>
        <TD>Pinkerton</TD>
    </TR>
    <TR>
        <TD>The Wolfgang Press</TD>
        <TD>Funky Little Demons</TD>
    </TR>
...
</BODY>
</HTML>
```

Figure 1: HTML encoded CD purchase order

provides one primarily stylistic way of representing your information. It is difficult to represent relationships between parts of your data, or to indicate the meaning of different parts of your data.

By contrast, XML allows the developer to create new markup symbols, called *tags*, to distinguish sections of a document. This means that each developer can create a set of custom tags that map directly to entities in the domain of their work, eliminating ambiguity in the document. This is one of the benefits of XML: it is flexible.

As an example, a purchase order of compact discs in HTML might look like the document in Figure 1. To a human reading this document, the meaning of each section of text is reasonably clear. But a computer reading this document can do little more than render the text according to a predefined set of rules. The computer cannot, for example, easily convert this textual representation into an online database.

However, the same document in XML might look like Figure 2. Now the document is easily readable by a computer. As a side benefit, the document is also easier for people to read. There is no longer any ambiguity about what the document represents, nor what each segment of text within the document means. The relationship between the pieces of data is perfectly clear. This is another important benefit of XML: it provides a clear, easy-to-understand format for data.

```
<PurchaseOrder>
    <ShippingAddress>
        <Name>XYZ Corporation</Name>
        <Street>2867 Coast Avenue</Street>
        <City>Mountain View</City>
        <State>CA</State>
        <ZIP>94043</ZIP>
    </ShippingAddress>
    <CD>
        <Artist>Weezer</Artist>
        <Title>Pinkerton</Title>
    </CD>
    <CD>
        <Artist>The Wolfgang Press</Artist>
        <Title>Funky Little Demons</Title>
    </CD>
    ...
</Purchase Order>
```

Figure 2: XML encoded CD purchase order

Because XML is an open specification, the software required to read XML documents is readily available from many sources. Several free packages are available for reading the documents, such as `expat` [4]. This means that developers now have an easy way to share data between programs: send it out as an XML document. Developers no longer need complex binary encodings for their data, nor are they dependent on proprietary data sharing systems. This is another benefit of XML: it is an open standard.

Of course, the ability to share data requires that developers agree on a particular set of tags to use for their data. If every developer uses their own set of tags, then the world will be little better off than when HTML was the only practical markup language. Fortunately, standardization efforts are already well underway in many domains. Those efforts will ensure that for many common applications, a set of XML tags already exists.

The primary advantages of XML, therefore, are its flexibility, legibility and its openness. It provides an easy to use and understand format for storing and sharing data.

2.1 Business-to-business applications with XML

Over the next few years, one of XML's uses will be as a data representation for business-to-business applications. In such applications, servers in one company communicate directly with servers in another company to automate business processes. For example, the inventory management system of a company might send an electronic purchase order to the sales system of a supplier in order to replenish inventory without any human intervention.

This is not a revolutionary idea; systems already exist that allow the automation of many business processes. However, these systems are based on complex binary encodings of data, such as EDI. The software to decode and process these encodings can cost millions of dollars to deploy within a company. And as with any binary data encoding, it is difficult to extend or customize. XML and the Internet make it easier and less expensive to implement business-to-business applications. They make trading communities possible, in which any company can easily and inexpensively communicate with any other company in the community.

3 Scriptics Connect

For all the potential that XML has to impact the business-to-business world and others, there are some significant problems that must be addressed. The worst of these problems is the sheer difficulty of programming XML applications. Presently, most software available for processing XML consists of low-level XML parsers. Most of these parsers require the programmer to work in a systems programming language like C or Java. As we will show later, using these low-level parsers is hard at best.

A second problem is the lack of infrastructure for XML applications. There is no standard mechanism for transmitting XML documents between entities. And there is no standard for integrating XML applications with other data sources, such as corporate databases or legacy applications.

Because XML is only a data representation, it has no ready solutions for these problems. With Scriptics Connect, our goal is to provide the missing infrastructure needed to make creating XML-based applications simple. The infrastructure components we provide include:

- *transport*: a means for communicating XML documents over the Internet and within an enterprise
- *integration*: a means for moving data between XML documents and other applications

Scriptics Connect combines Tcl and standard Web servers like Apache and Microsoft Internet Information Server with XML to provide this infrastructure. In addition, it provides two levels of programming abstraction to make creating XML applications "falling-over easy."

As shown in Figure 3, a Scriptics Connect installation consists of a World Wide Web server coupled with an XML parsing engine and one or more *document handlers*. A document handler is a Tcl script that describes how to process a particular type of XML document. Document handlers utilize a number of *connection points* to interface with various external applications. The XML parser and connection points are implemented as extensions to Tcl, so Scriptics Connect is essentially the combination of a Web server with an XML-enabled Tcl interpreter.

Scriptics Connect also includes the GUI applications Scriptics Connect Author, which allows the user to easily create document handlers, and Scriptics Connect Debugger, which allows users to debug document handlers installed on their server.

3.1 Scriptics Connect Server

The Scriptics Connect server consists of a Web server coupled with a Tcl interpreter. We chose to support several different popular Web servers in order to make it easier for Scriptics Connect to integrate into a company's existing Web infrastructure.

Building Scriptics Connect on top of a Web server gave us a good answer to one of the infrastructure problems we wanted to address: transport. Using a Web server made it easy to use HTTP as a transport protocol for XML. HTTP is a natural choice for this task for several reasons. First, it is an existing open standard. Second, most corporate firewalls are already configured to pass HTTP requests. One of the biggest problems with existing business-to-business solutions is the use of proprietary transports that require complicated and expensive integration efforts.

The Tcl interpreter in the Scriptics Connect server has been augmented with several extensions. First, it includes extensions for processing and generating XML documents:

- `tclExpat`: a fast, non-validating XML parser based on expat [3, 4]
- `tclDomPro`: an XML Document Object Model interface built on top of tclExpat
- `xmlact`: an API for binding Tcl scripts to parts of an XML document
- `xmlgen`: an API for generating XML documents

Second, it includes several connection points for interfacing with various data sources:

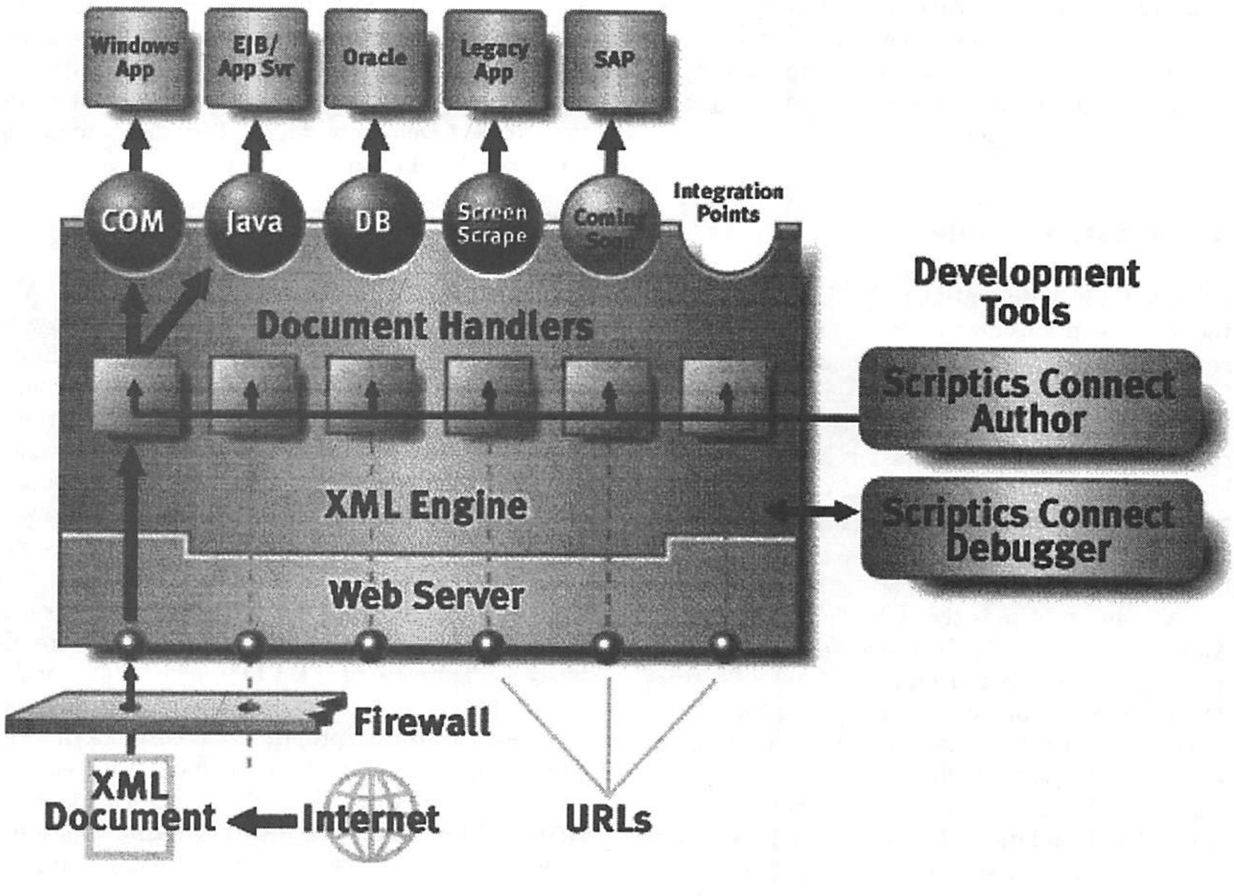


Figure 3: Components in the Scriptics Connect architecture

- `tclCom`: allows COM objects to be created and invoked from Tcl; developed at Scriptics but influenced by the open source tCom extension by Chin Huang [5]
- `TclBlend`: provides access to Java classes, objects, Enterprise JavaBeans and databases, via JDBC [9]
- `OraTcl`: provides access to Oracle databases [8]
- `Expect`: allows Tcl scripts to communicate with and automate applications that normally expect to be interacting with a user typing at a terminal [6]

Tcl is a natural choice for implementing XML applications. Because XML is a text-based format, it is important to have powerful and easy-to-use string and regular expression operations. Tcl's "everything is a string" model is a good fit for manipulating XML data. Additionally, XML is represented using the Unicode character set. As of version 8.1, Tcl uses Unicode as its native character set, which makes manipulating XML data simple. Finally, Tcl has an extensive collection of extensions freely available on the Internet. Many of the connection points in Scriptics Connect come from open source extensions. This allowed us to provide substantially more functionality in our first release than would have been possible if we had implemented everything ourselves.

3.2 XML Document Processing

The combination of a Web server plus the Tcl platform addresses many of the transport and integration issues surrounding XML. However one of the remaining issues we had to deal with was the difficulty of processing XML documents. One of our goals for Scriptics Connect was to raise the level of programming, both to reduce the amount of code that must be written and to reduce the programming skills required to develop business applications.

Most tools for processing XML documents today consist of low-level XML parsers. To use these tools, a programmer must write code in a system programming language like C or Java. These parsers fall into one of two groups: event-based and tree-based. Event-based parsers treat XML documents as streams of data and generate events for each opening and closing tag in the document. Event-based parsers require the user to manage state between callbacks in order to gather data for processing. On the other hand, tree-based parsers treat XML documents as data structures and build in memory a tree representation of an XML document. Using a tree-based parser typically involves writing code that uses Document Object Model (DOM) [1] interfaces to traverse the nodes in the resulting tree to find relevant data. Both models are cumbersome and involve a lot of programming to perform even simple tasks.

As an example, consider the sample XML encoded CD purchase order shown in Figure 2. Suppose a developer wants to import the information in that XML document into a local relational database. Using just the event-based XML parser **`expat`**, a programmer would have to write C code similar to that in Figure 4. The complexity of extracting the data overwhelms the original programming task of inserting the data into the database. Real XML documents are even more complex than our sample purchase order. This makes XML programming a prime candidate for an abstraction layer.

3.3 The Post-itⓡ Model

Instead of using a strictly tree or event based model, we chose to combine them into a hybrid approach. The metaphor we used to simplify XML programming is that of attaching Post-it® notes to a paper document. Imagine that instead of an XML document, you have a physical copy of the type of document you need to process. If one person were to describe to another person how to process a paper form, they might do it by taking a copy of the form and placing Post-its® on the form, as shown in Figure 5. Each Post-it® would have instructions for processing a particular part of the form, and it might indicate fields from the form that are needed to carry out the instructions. Once a form has been thusly annotated, it could be given to a person who could then carry out the instructions on similar forms as they arrive.

With this abstraction, we can effectively reduce the task of parsing XML to the task of placing Post-it® notes on a document. Pseudo code exploiting this abstraction might look like this:

```
at the CD element call
    addToDatabase with Artist, Title
```

The developer doesn't worry about the technical details of parsing the XML data or finding a position within the document tree. Instead they can focus on the problem they are trying to solve. An important side benefit of our Post-it® abstraction is that it is readily understandable even by non-engineers.

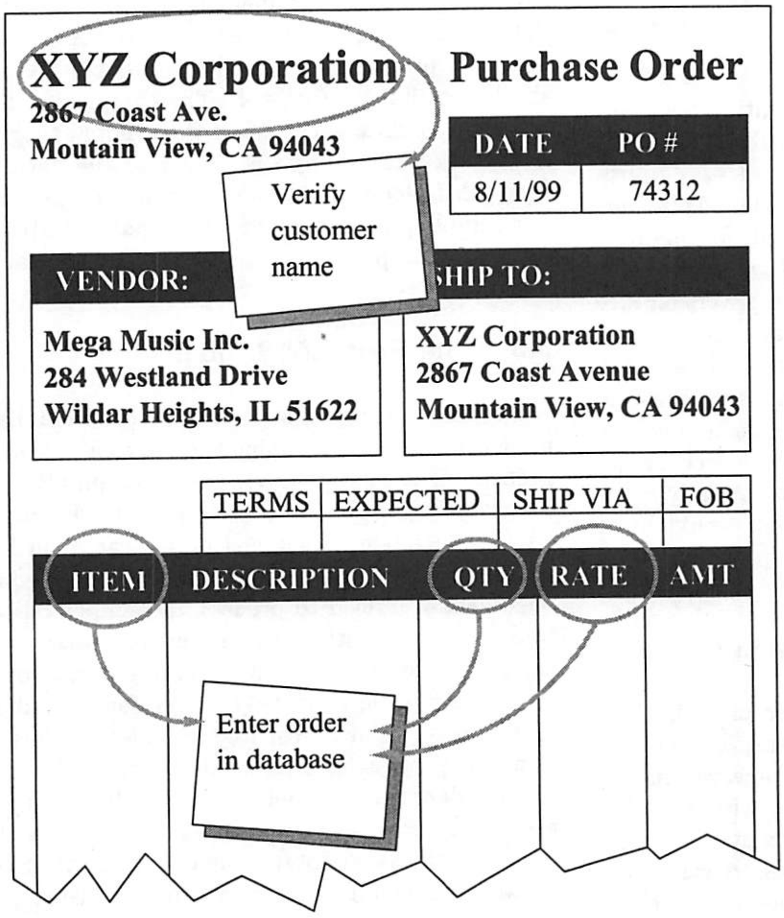


Figure 5: The Post-it® metaphor

```
char currentArtist[128];
char currentTitle[128];
int inArtist, inTitle;
void startElement(void *userData,
                  const char *name,
                  const char **atts)
{
    inArtist =
        (strcmp(name, "Artist") == 0) ? 1 : 0;
    inTitle =
        (strcmp(name, "Title") == 0) ? 1 : 0;
}
void endElement(void *userData,
                const char *name)
{
    if (strcmp(name, "CD") == 0) {
        addToDatabase(currentArtist,
            currentTitle);
        return;
    } else if (strcmp(name, "Artist") == 0) {
        inArtist = 0;
    } else if (strcmp(name, "Title") == 0) {
        inTitle = 0;
    }
    return;
}
void cdataHandler(void *userData,
                  const char *s,
                  int len)
{
    if (inArtist) {
        strncpy(currentArtist, s, len);
        currentArtist[len] = '\0';
    } else if (inTitle) {
        strncpy(currentTitle, s, len);
        currentTitle[len] = '\0';
    }
}
void main() {
    ...
    XML_SetElementHandler(parser,
        startElement, endElement);
    XML_SetCharacterDataHandler(parser,
        cdataHandler);
    ...
}
```

Figure 4: C code for processing an XML encoded CD database

3.4 xmlact: the XML Action API

Our implementation of the Post-it® abstraction took the form of a new API called xmlact. The xmlact API consists of a few commands, `parserCreate`, `parse`, and `parserDelete` for creating, using, and deleting an XML parser. In addition, it has one command, `action`, which is used to associate a Tcl script with a location in an XML document. Locations in an XML document are specified using a path syntax where nested tags are separated by slashes, much like a file name: `PurchaseOrder/CD/Artist`. This is similar to, but simpler and less complete than, the syntax used by XPath [2], a W3C Recommendation for addressing parts of an XML document. These paths are really patterns that are compared with each location in a document to find a match. The parser walks the XML document looking for locations that match a pattern. When a match is found, the corresponding Tcl script is invoked.

Although this basic pattern matching mechanism is a very easy interface that greatly simplifies the task of invoking a Tcl script at a particular place in the document, it doesn't make the task of collecting related pieces of data all that much easier. In order to extract data from the document, the developer would still need to use a mechanism similar to that employed in the earlier C example in Figure 4 to pass data from one action to the next. To simplify the data collection task, we added the ability to refer to data that is contained within the current tag.

Using the xmlact `action` command, the sample C code in Figure 4 can be replaced with the following simple Tcl code:

```
xmlact::action parser CD addToDatabase \
-text Artist -text Title
```

When the XML encoded CD database in Figure 2 is processed, the following sequence of procedure calls will result:

```
addToDatabase Weezer Pinkerton
addToDatabase {The Wolfgang Press} \
{Funky Little Demons}
...
```

A specifier like `-text Artist` indicates that the second word of the Tcl command should consist of the text in the `Artist` subelement of the `CD` element. Several types of specifiers exist, for extracting different parts of the XML document:

- `-text` *tagpath*: retrieve the contents of the specified subelement as text with no embedded tags

- `-value` *tagpath attribute*: retrieve the value of the given XML *attribute* for the specified subelement
- `-attributes` *tagpath*: retrieve a Tcl list of XML attributes and values for the specified subelement
- `-tag`: Retrieve the name of the triggering element
- `-path`: Retrieve the complete path of the triggering element
- `-literal` *string*: Pass the literal *string* through as an argument to the command

These specifiers allow the developer to easily access most pieces of data in an XML document. We deliberately placed a few restrictions on the data available in order to improve performance and reduce the amount of memory required during parsing of large documents. In particular, an action can only refer to data contained within the element it is attached to. Also, we ignore some of the more esoteric XML features like processing instructions. We expect that most business to business applications will not use these features, so we decided to target the "sweet-spot" in order to reduce the complexity of the interface.

3.5 xmlgen: The XML Generation API

There are at least two situations that require some sort of outgoing XML. First, it may be necessary to return data as the result of an incoming request. XML is a natural choice for formatting that response. Second, users of Scriptics Connect may wish to initiate requests as well as respond to them. In order to address these needs, we created the xmlgen API, a means for creating XML on-the-fly.

The xmlgen API has two distinct interfaces for creating XML. To illustrate the difference between the two, consider the following sample XML:

```
<response>
    <header>
        Some text
    </header>
    <body>
        Some more text
    </body>
</response>
```

The first interface is similar to that used in Don Libes `cgi.tcl` [7], in which the document is constructed by a series of nested function calls. The following code uses this interface to create the sample XML shown:

```
xmlgen::element mydoc response {
    xmlgen::element mydoc header {
        xmlgen::text mydoc "Some text"
    }
    xmlgen::element mydoc body {
        xmlgen::text mydoc "Some more text"
    }
}
```

In this style, each call to `xmlgen::element` produces an opening and closing tag with the specified name. The API will execute whatever code is in the body of the call between writing out the opening and closing tags. This style of XML document creation has a couple of benefits: it is easy to see the mapping between the code producing the document and the resulting output, and it works well when the entire document can be produced at once.

However, in some cases, this model does not work well. For example, if you need to incrementally produce the document as you process something, it would be easier to use a more streaming model. In this case, the second interface is useful. This interface uses two commands, `xmlgen::startElement` and `xmlgen::endElement` to create opening and closing tags for elements in the document. The following code uses the streaming interface to create the same sample XML:

```
xmlgen::startElement mydoc response
xmlgen::startElement mydoc header
xmlgen::text mydoc "Some text"
xmlgen::endElement mydoc
xmlgen::startElement mydoc body
xmlgen::text mydoc "Some more text"
xmlgen::endElement mydoc
xmlgen::endElement mydoc
```

One disadvantage of this interface is that it is harder to see the mapping between the code and the resulting document. In addition, the programmer is responsible for maintaining balanced tags, unlike the first interface. However, in some cases, the streaming output model is simply easier to work with.

3.6 Scriptics Connect Author

Although the xmlact interface makes parsing tasks much easier than if users had to write directly to low-level parsing APIs, there is still a lot of coding involved in performing the various integration tasks facing an XML application developer. For example, it is still troublesome to write the code needed to access a database. To address this issue, we developed Scriptics Connect Author, a GUI tool to assist in creating document handlers.

Author provides two primary benefits. First, it provides a simple graphical interface that helps visualize the task of attaching actions to parts of an XML document. Author displays a schematic view of an XML document; the user selects a particular XML element and associates an action with it. This task effectively generates a call to `xmlact::action`, and is really just a "glossy cover" for the `xmlact::action` function. Figure 6 shows a screenshot of this interface.

Second, through the use of various *wizards*, Author provides easy access to the connection points in Scriptics Connect. Having associated an action with an XML element, the user must define what behavior that action should have. One way to do this is to write a segment of Tcl code. This provides a great deal of flexibility, because the developer can write arbitrary Tcl code. However, we realized that in most cases, the developer does not need that degree of flexibility, and would be better served by an interface customized to a particular type of action. Thus we created a set of action wizards, each of which is optimized for a different type of action, and provides a graphical interface that makes that kind of action easy to implement. For example, we have created a wizard for inserting data into a database; Figure 7 shows a screenshot of this wizard.

We have created several different wizards for Author, including:

- Database wizards for inserting, deleting, updating, and querying
- A condtional wizard for controlling the execution of actions at runtime based on the data in an XML document
- A Tcl variable wizard for extracting data values into Tcl variables for later use
- A TclScript wizard, which allows the developer to supply arbitrary Tcl code, in case our premade wizards are inadequate for their needs

In addition, the Author Wizard API is open, so that users can create new wizards that are tailored to their particular needs.

4 An End-to-End Example

An end-to-end example of how XML can help to automate business-to-business applications begins with two companies, NewCo and OfficeStuff, that do business with each other. NewCo, being a new and rapidly growing company, finds that it often needs to purchase office supplies from OfficeStuff.

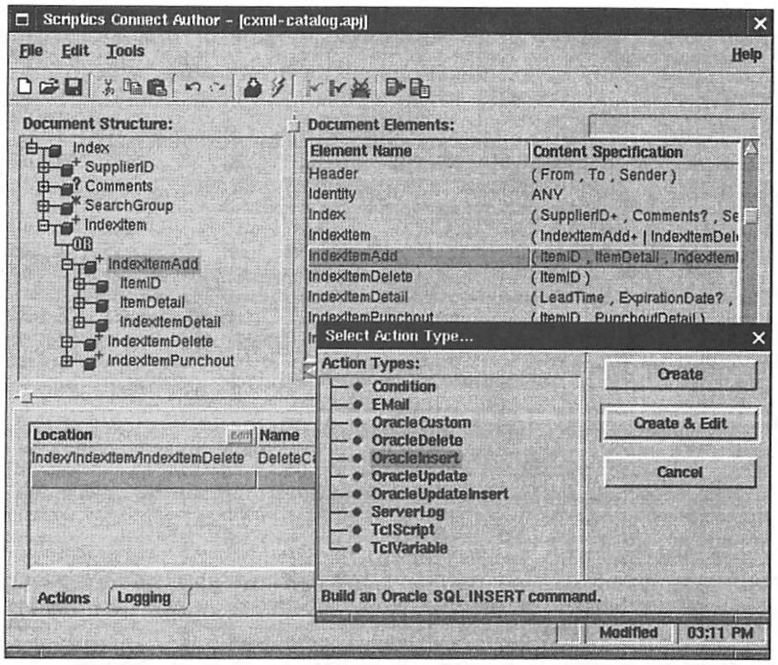
Initially, the supply chain proceeds as follows:

1. Employees at NewCo fill out an internal form requesting supplies and send it to the VP of Buying.
2. NewCo's VP of Buying calls her secretary and asks him to order some office supplies.
3. The secretary writes up a purchase order and faxes it to OfficeStuff headquarters.
4. Sales Agent Sal at OfficeStuff receives the purchase order, verifies that NewCo is a real customer and has enough money for the order, then calls the shipping department.
5. An inventory manager at OfficeStuff gathers the items needed to fill the order, updates the inventory database, and arranges for the items to be shipped to NewCo.
6. Sal sends a bill to NewCo.

This system is perfectly functional, but it is time consuming, costly and inefficient. Several people are involved in the chain, each of which increases the cost and chance for error. But the companies could overhaul their supply chain with XML and set up an automatic supply chain, eliminating the need for the extra "middle-men," reducing costs and increasing efficiency and accuracy.

In a partially XML-enabled scenario, NewCo and OfficeStuff, recognizing the inefficiency of their system, get together and agree on the format for the XML documents they will use to automate the supply chain. In this case, there are two: a purchase order and a bill. Now, NewCo can set up an internal purchasing system whereby employees can request office supplies; the orders are collected and formatted as a single XML request, which is then sent via email to Sales Agent Sal at OfficeStuff. Sal, upon receiving the XML document, prints it and processes it much as before. At this stage, half of the supply chain is automated. This is good for NewCo, but OfficeStuff has made no benefit yet. Thus the partially XML-enabled supply chain proceeds as follows:

1. Employees at NewCo log in to the purchase system and request supplies.
2. The purchase system collects requests and sends a purchase order to OfficeStuff.



Element Name	Content Specification
Header	(From , To , Sender)
Identity	ANY
Index	(SupplierID+ , Comments? , Se
IndexItem	(IndexItemAdd+ | IndexItemDel
IndexItemAdd	(ItemID , ItemDetail , IndexItem
IndexItemDelete	(ItemID)
IndexItemDetail	(LeadTime , ExpirationDate? ,

Location	Name
Index/IndexItem/IndexItemDelete	DeleteC

Build an Oracle SQL INSERT command.

Figure 6: Screenshot of the Author interface

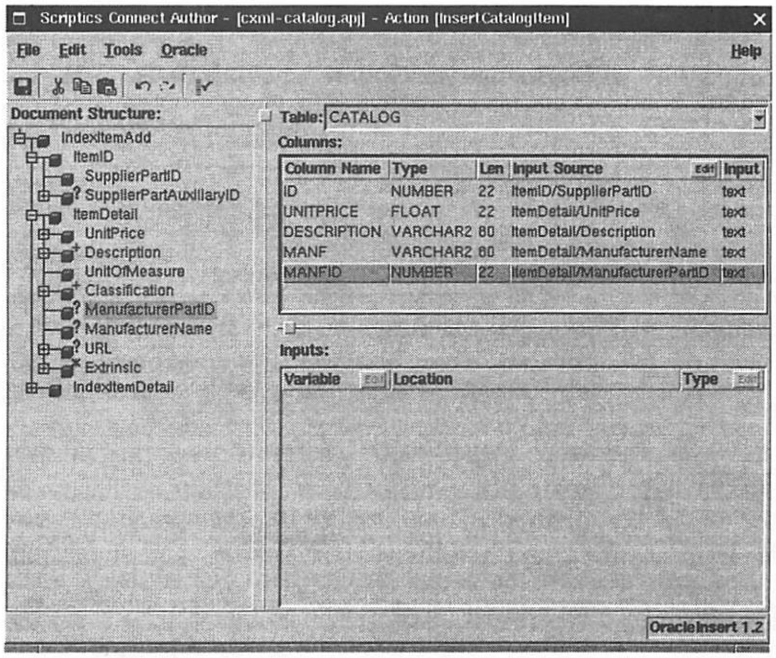


Table: CATALOG

Columns:

Column Name	Type	Len	Input Source	Input
ID	NUMBER	22	ItemID/SupplierPartID	text
UNITPRICE	FLOAT	22	ItemDetail/UnitPrice	text
DESCRIPTION	VARCHAR2	80	ItemDetail/Description	text
MANF	VARCHAR2	80	ItemDetail/ManufacturerName	text
MANFID	NUMBER	22	ItemDetail/ManufacturerPartID	text

Figure 7: Screenshot of a database insert wizard. The user selects a database table, then drags XML elements and attributes from the tree on the left into the "Columns" listbox on the right to specify which parts of the document to use to fill the database. In addition, the user can drag things to the "Inputs" listbox, to allow simple manipulation of the data before inserting it into the database.

3. Sales Agent Sal at OfficeStuff receives the purchase order, verifies that NewCo is a real customer and has enough money for the order, then calls the shipping department.

4. An inventory manager at OfficeStuff gathers the items needed to fill the order, updates the inventory database, and arranges for the items to be shipped to NewCo.

5. Sal sends a bill to NewCo.

With a system like Scriptics Connect, OfficeStuff can automate their half of the supply chain as well. They can create a document handler to process the XML-based purchase orders from NewCo. This document handler might connect to the OfficeStuff customer database to verify the validity of the customer and the depth of the customer's pocketbook. It could connect to the inventory database, verify that sufficient inventory was available, update the inventory, and issue an order to the inventory manager to load the furniture on the delivery truck. It could even issue an appropriate bill to NewCo for the purchase.

After creating the document handler, OfficeStuff can publish it to their Scriptics Connect server and notify NewCo of the URL to use when sending purchase orders. After NewCo has adjusted their internal purchase system to use HTTP instead of email for delivering the purchase order, the supply chain will be as automated as it can be, unless some sort of machine is used to load the delivery truck. The system now proceeds as follows:

1. Employees at NewCo log in to the purchase system and request supplies.

2. The purchase system collects requests and sends a purchase order to OfficeStuff.

3. OfficeStuff's Scriptics Connect server verifies the validity of the customer, checks the customer's credit line, checks and updates the inventory database, sends a shipping request to the shipping department and bills NewCo.

4. An inventory manager at OfficeStuff gathers the items needed to fill the order and arranges for the items to be shipped to NewCo.

Instead of the several people involved in the original purchase, only the bare minimum people required are involved. The savings to both companies in this case is probably not tremendous, but it probably is noticeable. And OfficeStuff can now turn to its other customers and ask them to use XML as well, further reducing their operating costs. Similarly, NewCo can ask its other suppliers to use XML-based systems. As more and more entities become XML-enabled, those companies that are automated will enjoy further reduced costs.

5 Problems with Scriptics Connect

Although we feel we have been successful in achieving our initial goals, we recognize that there are some problems and shortcomings with our implementation.

One obvious shortcoming is in the area of generating XML. Our system works very well for receiving XML, but it is difficult to generate XML for transmission. The xmlgen API is completely functional, but it is not very user-friendly. We would like to provide graphical facilities for describing how to generate XML. Ideally, such a GUI would provide the same simplicity that Author provides for handling incoming documents.

Another area that needs improvement is data collection. Our current facilities for gathering data from an XML document are complete, but they are difficult to apply to some situations. For example, retrieving the data from all of the instances of a repeated element in an XML document is not straightforward. The user must create a TclScript action and append values to a list. Complex data structures are another example. Presently, the user can extract the data, but must do so one field at a time, which can be tedious. In both cases, we believe that we can create new mechanisms to address the problem.

A third area is code and action reuse. It is currently not possible to share actions between document handlers. It is easy to imagine scenarios that would call for the sharing of actions, making this a potentially large inconvenience for our users.

6 Future Work

We have an aggressive development schedule set for Scriptics Connect. In version 2.0, we plan to add several features:

- Built-in integration with online trading communities like AribaNet and CommerceOne.
- Additional transport mechanisms for receiving and sending XML, such as FTP and SMTP
- Better monitoring tools

In version 3.0, we plan to further enhance the GUI, perhaps integrating Author into an IDE for working with Scriptics Connect.

7 Conclusions

XML has the potential to greatly reduce operating costs and increase efficiency and accuracy in many business-to-business applications. However, the lack of infrastructure and the difficulty of creating XML applications are two significant barriers to the spread of XML. We had two goals in mind when creating Scriptics Connect:

- Provide the missing infrastructure needed to enable XML-based business-to-business applications
- Simplify the process of creating XML-based business-to-business applications

We achieved the first goal through the use of standard World Wide Web servers, to provide a transport mechanism, and Tcl, to provide an integration mechanism. We achieved the second goal with the xmlact API and the Scriptics Connect Author. Tcl proved to be an ideal choice for XML-based business-to-business applications, because of its ability to handle XML well, and because of its ability to connect to many external applications.

8 Acknowledgements

We would like to thank the employees of Scriptics Corporation, without whom Scriptics Connect could not have been created. In addition, we would like to thank the authors of the open-source software packages that we made use of in Scriptics Connect.

Post-it® is a registered trademark of 3M.

References

[1] Document Object Model: http://www.w3c.org/DOM. World Wide Web Consortium.

[2] XPath: http://www.w3c.org/TR/xpath.html. World Wide Web Consortium.

[3] Steve Ball. XML Support for Tcl. In *Proceedings of the 6th Annual Tcl/Tk Workshop*, page 109. USENIX, September 1998.

[4] James Clark. Expat Home Page: http://www.jclark.com/xml/expat.html.

[5] Chin Huang. tCOM: http://www.vex.net/~cthuang/tcom/.

[6] Don Libes. *Exploring Expect.* O'Reilly & Associates, Inc., 1994.

[7] Don Libes. Writing CGI Scriptics in Tcl. In *Proceedings of the 4th Annual Tcl/Tk Workshop '96*, pages 189–201. USENIX, July 1996.

[8] Tom Poindexter. OraTcl: http://www.nyx.net/~tpoindex/tcl.html.

[9] Scott Stanton. TclBlend: Blending Tcl and Java. *Dr. Dobb's Journal*, February 1998.

Supporting Information Awareness Using Animated Widgets

D. Scott McCrickard Q. Alex Zhao
Graphics, Visualization, and Usability Center
and College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{mccricks,azhao}@cc.gatech.edu

Abstract

This paper describes a Tcl/Tk widget set extension that supports three animated behaviors: fading, tickering, and rolling. This extension is targeted for use in information awareness applications that monitor and communicate constantly changing information. Described is the programming interface for the widgets as well as the design decisions made in creating them and programs that use them. Introduced in the description are two new techniques called automatic markups and history-based shadowing, highlighting techniques used to identify and communicate the nature of the changes.

1 Introduction

What is animation? Baecker and Small describe it as "sequences of static images changing rapidly enough to create the illusion of a continuously changing picture" [1]. It has been used to generate emotion, to provide entertainment, and, of interest in this paper, to communicate information.

One domain where animation may prove useful is *information awareness*, the need to stay aware of changes to information while accomplishing other tasks. Consider the ever-expanding pool of constantly changing information that is readily available on the Internet and World Wide Web. Stock prices climb and fall, news bulletins arrive, email queues grow, ball teams score points, and the weather changes. While people care about this information, their primary focus is generally on more important tasks: editing documents, programming code, or other job-related activities.

We feel that gradual and repetitive animation can be used to create useful and usable information awareness applications. By animating large amounts of information in a small space, the remaining space can be used for other applications. Changes to the information can be integrated gradually into the display in the next iteration, minimizing the disturbance to the user. Constantly cycling through the entire information space lessens the number of physical interactions required to obtain the information – rather than having to press a series of buttons or keys to get information, the user need only wait for it to cycle through.

To provide easy creation and use of animated effects we have integrated animation support into Tcl/Tk via three widgets. We focused on three effects that seem to meet the criteria discussed earlier: the *fade* widget cycles between items of text, bitmaps, or images, the *ticker* widget scrolls information horizontally across the screen, the *roll* widget scrolls information vertically. Section 2 describes the widgets in more detail.

In creating an animated widget set, numerous design decisions had to be made, not the least of which is the choice of a language. The Web-aware nature of Tcl combined with its powerful string parsing capabilities makes it a good choice for information monitoring applications. The easily extensible widget set and well-defined programming interface of the Tk toolkit provide a good framework for the implementation of animated widgets. Section 3 outlines selected implementation details made in constructing animated widgets, including more reasons behind our choice of Tcl/Tk as the implementation language.

While animation has many potential benefits, it has several drawbacks as well. Animation is often seen as distracting or annoying because of its constant motion and rapid visual changes. Our widget set addresses this problem by giving both the programmer and the end-user significant power over the animation. Another problem with animation specific to awareness is that it can be difficult to see where changes occur and what the previous states of information was. By augmenting the display using information from previous states with techniques we call *automatic highlighting* and *history-based shadowing*, we attempt to lessen the effects of these problems. Section 4 describes in detail our solutions.

Our animated widgets have been available and in use for over a year. The animated widgets have been used by over fifty programmers in a variety of information awareness applications, and some of the programs have been used by more than a hundred users. Section 5 discusses some guidelines for writing information awareness applications based on feedback from both programmers and users.

Overall, we believe that animation is an important technique to have available in a user interface toolkit, and Tcl/Tk is an appropriate language in which to include animation. This paper describes the behavior of the Agentk animated widgets and explores some of the design decisions made in creating them.

2 Animated widgets in the Agentk toolkit

Animation has the potential to enhance the awareness capabilities of interfaces. Our Agentk toolkit contains several interface widgets intended to make it easy for programmers to design awareness applications and easy for people to use them in maintaining awareness of changing information.

Animated widgets are a subclass of *mega-widgets*, a collection of widgets that are operated using a single interface. For example, the LabelText mega-widget in the Tix library provides a single interface to a label widget and an entry widget [8]. Just as mega-widgets augment the power and ease of use of widgets by packaging several of them together as a single new widget, animated widgets augment the behavior of a widget by constantly changing its appearance at regular intervals.

Agentk is a widget toolkit designed to assist in the creation of agent-like programs such as the information awareness applications discussed earlier. Agentk contains three animated widgets: the *fade* widget, the *ticker* widget and the *roll* widget. The fade widget fades between multiple blocks of text, bitmaps, or graphical images. It can be used when the blocks of information have a fixed height and width. The ticker widget horizontally scrolls or "tickers" a stream of text. It can be used for variable length streams of text. The roll widget scrolls text vertically. It is well suited for ordered lists of items.

The programming interface for the fade, ticker, and roll widgets is identical to that for labels, scrollbars, or any other widget: the programmer specifies the widget creation command (ex. `label`, `scrollbar`, `fade`), the position in the display tree (`.fader`, `.dialog.ok`), and possibly some options (`-width 50`, `-fg red`, `-shadowhistory yes`). The options include all those of the standard label widget used to display static text and images, as well as additional options that allow a programmer to show the contents of more than one variable and to control aspects of the animation. In so doing, we provide a programming interface that can be easily understood by a Tcl/Tk programmer.

2.1 Fade widget

Rapid changes in the appearance of a widget often attracts the user's attention to the widget. While this is advantageous in many situations, it could be detrimental in an interface where the users' attention is primarily focused on other tasks. Instead, we need a

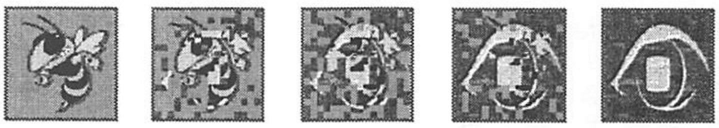


Figure 1: A time-lapse series of the fade widget for two images. Rather than perform compute-intensive calculations to achieve a fading effect, the original image is broken into pieces, and the pieces of the original are gradually replaced with pieces of the final.

widget that can change continuously to match the large and dynamic information space but will change gradually to avoid interrupting the everyday tasks of the user.

The fade widget displays multiple blocks of either text, bitmaps, or graphical images within a given space by gradually fading between them. We expect that the gradual change will be less distracting than a sudden switch yet will allow multiple information blocks to be displayed in a single area. The speed with which the fade occurs can vary depending on the nature of the application: if the widget is designed to attract attention and can be easily stopped by the user, a quick fade might be used, while a secondary display designed to run all the time might call for a slower, less intrusive fade.

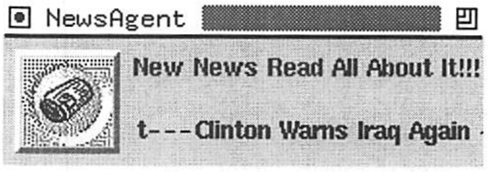
The following lines of code give an example of the fade widget displayed in Figure 1.

```
set i1 [image create photo -file buzz.gif]
set i2 [image create photo -file eye.gif]
fade .f -width 100 -height 100 \
  -imagevariable [list i1 i2]
pack .f
.f run
```

The first two lines create two images that are to be faded (actually stippled) in and out. If the values of the variables are changed later, the new images will fade in on the next iteration. The third line creates the widget to display the information contained in the variables `i1` and `i2`. Unlike the textvariable option for other widgets, the fade widget supports a list of variables, fading between each in the list.

2.2 Ticker widget

The ticker widget provides a ticker-tape-style display that scrolls or "tickers" text across the screen. As with the fade widget, a gradual



NewsAgent

New News Read All About It!!!

t---Clinton Warns Iraq Again ·

Figure 2: A news agent that incorporates a fading widget (top right) and tickering widget (bottom right) to alert the user of new stories.

tickering can be less distracting than a sudden switch. The ticker widget has the added advantage that a stream of text could be any length since it never needs to be displayed in its entirety at any given time.

In addition, we take advantage of the natural reaction to grab and pull the ticker widget to make it slow down, stop, back up, or move. When users click and hold down the mouse button within the widget, they can manipulate the information to move in the desired direction and speed. By providing this functionality, the user has more control over the way in which the information is displayed.

The following lines of code gives an example of the ticker widget.

```
ticker .t -width 100 \
  -textvariable [list u1 u2 u3]
pack .t
.t run
```

The first line creates a 100-character-wide widget that is configured to display the information in the variables `u1`, `u2`, and `u3`. The second line packs the ticker widget just as any other widget would be packed, and the third line begins the tickering process.

The ticker widgets have been used in applications to show article headers which, unlike sports scores or stock quotes, can vary widely in length. Figure 2 shows an interface where the ticker widget has been used.

2.3 Roll widget

The roll widget "rolls" or vertically scrolls text across the screen. It seems best suited for a list of items or a columns of information where the order and position of the information is important. The ordering of the list and the positioning of the columns is more apparent in the roll widget than in a ticker or fade widget. As with the ticker widget, the user can grab, hold, and move the roll widget to alter the visible information.

Consider the following quick-and-dirty example for monitoring a printer queue on a Unix system.

```
roll .r -width 60 -height 6 \
   -justify left -font fixed
pack .r
.r run
proc checkq {} {
  .r configure -text [exec lpq]
  after 20000 checkq
}
checkq
```

The first line creates a roll widget that is six lines high and 60 characters wide. The text is left justified and the font is fixed to ensure that the columns will line up. The second and third lines pack the widget and start it running. The `checkq` procedure repeatedly checks the contents of the printer queue using the Unix lpq command and configures the roll widget to reflect the results of this command. Figure 3 shows the result of this script.

System administrators could use this short script to keep an eye on buggy printers or to watch for printer misuse. A user who is printing many documents over a short period of time could extend the script to watch several printers at once to know at any time which is the least busy.

2.4 Options, subcommands, and bindings

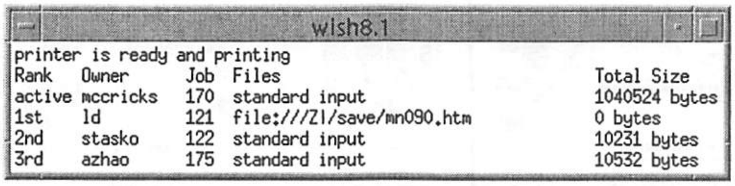
This section describes the subcommands, options, and bindings for the Agentk animated widgets.

A subcommand is a command that is available within a widget command. For example, all Tcl/Tk widgets (including the animated widgets) support the `configure` subcommand for querying and altering the widget options. Animated widgets additionally support the following subcommands:

- `run` command starts the animation effect for the widget.
- `pause` command pauses the animation until the run command is reissued.
- `jump` (fade only) jumps to the next item in the fade display list.

A configuration option alters the behavior and appearance of a widget. Common options include `-geometry`, `-font`, and `-width`. The following are some selected options that are available with the animated widgets.

- `-speed` indicates the speed with which the animation runs.
- `-delay` (fade only) is the delay in milliseconds before an item that has faded in begins to fade out.
- `-text[variable]`, `-bitmap[variable]`, and `-image[variable]` control the information that appears in the widget. When the value is changed, the new information animates into view as the old information disappears. For the variable options (such as `-textvariable`), the contents of each variable are animated on the screen. Only one of these options can be used for any given widget (the variable options have highest precedence, and images take precedence over bitmaps, and bitmaps over text).
- `-separator` contains a sequence of characters that separate entries in the ticker and roll widgets. This option is overridden by `-separatorbitmap` or `-separatorimage`, which can contain a bitmap or image separator.
- `-markupstyle` indicates a typeface markup (either bold, italic, or a color) used to highlight changes in information. Thus, if new text appears and the



```
wish8.1
printer is ready and printing
Rank   Owner     Job Files                                      Total Size
active mccricks  170 standard input                             1040524 bytes
1st    ld        121 file:///Z|/save/mn090.htm                  0 bytes
2nd    stasko    122 standard input                             10231 bytes
3rd    azhao     175 standard input                             10532 bytes
```

Figure 3: A roll widget that displays the contents of a printer queue. The information is rolled vertically across the screen. The user can grab and move the display if desired and can throw (drag and release) it to adjust the speed.

markupstyle is set to red, the new text will be red. To end the highlighting, `-markupcount` can be set to a number of iterations or `-markuptimeout` can be set to a length of time after which the markups will disappear. Section 4.1 talks more about automatic markups.

- `-shadowhistory` (a boolean) indicates whether history-based shadowing should be enabled. If it is, previous states of the displayed information (text only) are shown using a shadow effect. See Figure 5 for an example. `-shadowcount` and `-shadowtimeout` can be used to remove the shadowing after some number of iterations or some length of time. Section 4.2 talks more about history-based shadowing.

- `-throw` (a boolean) indicates whether speed should be adjusted when user "throws" a ticker or roll widget; that is, drags and releases it.

- `-drive` synchronizes widgets of the same type so that they will run in lock step. For example, the two fade widgets in Figure 4 must always run together as one widget shows images and the other shows labels for the images. As one drives the other, each step in the two fades will be calculated and displayed at the same time.

Bindings define the behavior when certain actions are performed by a user on a widget. Below are the default bindings. A programmer can tailor the bindings based on the needs of their applications.

- `ButtonPress` binding stops the animation effect. Users who want to read the text without being distracted by the animation can press and hold the mouse button.

- `Motion` of the mouse while the mouse button is being held down drags the ticker and roll widgets.

- `ButtonRelease` for the fade widget jumps to the next block of information. For the ticker and roll widget, it calculates a new speed based on the speed at which the widget was dragged. For all widgets it restarts the animation effect.

2.5 Applications using animated widgets

Animated widgets have been used in a variety of programs by over fifty programmers. The figures in this paper show some of the interfaces they have designed. This section focuses on a few interfaces, all of which are available from the Agentk home page given in the Conclusions section.

The NewsAgent interface repeatedly downloads and parses a news Web page looking for new articles (see Figure 2). It alerts users of new articles by changing the color of an image and rapidly fading in and out a "New News" message. The interface contains a ticker widget that continually tickers through the news headers. If a user sees an article of interest, by pressing the image a news viewer can be raised and the user can read the article in its entirety.

The tkscore interface monitors college basketball scores and displays them using a user-selected animated widget or using a standard label widget. We introduced this tool during the NCAA Basketball Tournament, a season ending single-elimination tournament

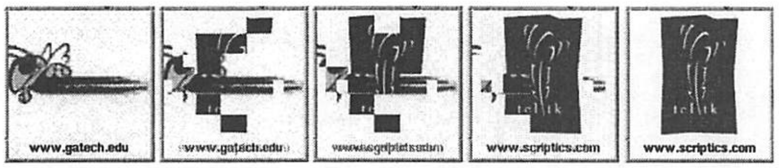


Figure 4: The fade interface of the CWIC passive Web browser. The display consists of two fade widgets running in sync using the -drive option. The upper one fades between images and the lower one fades between text labels for the image. The first frame shows an initial image and text. The next two frames show how the image and text fade away as the new ones fade in. The final two frames show how the new image and text appear in their entirety as the old disappear.

that generates significant interest and excitement. We asked the users to complete a questionnaire on their informational and animation preferences. The results of this study (available in [9]) have been used to further the development of our animated widgets and have led to the development of the tkwatch interface, a more general tool for monitoring stock quotes, news headlines, weather data, personal information, and sports scores.

The CWIC passive browsing system (see Figure 4) continually browses selected Web sites, identifying key images and presenting them to the user by gradually fading between images. Note that the URL where each image was found is provided as well so that the user can visit the page if an image of interest appears. CWIC is described in detail in [4].

3 Incorporating animation into Tcl/Tk widgets

The style of programming represented by scripting languages is well-suited for information awareness applications. As noted by Tcl creator John Ousterhout, scripting languages are designed to "glue" together existing resources making them easier and more efficient to use [13]. As this closely matches the purpose of awareness applications, (to act as an intermediary to the monitoring of information resources that otherwise may be tedious and time-consuming to do), it seems that scripting languages are an appropriate choice for authoring awareness applications. Tcl/Tk in particular seems to be a good choice for awareness applications. Tcl is designed to be a platform-independent scripting language with an extensible graphical toolkit (Tk) for interface design [12]. Extending this toolkit to include animated widgets provides a solid programming platform for the implementation of awareness applications.

To maintain consistency and decrease the learning time for programmers, animated widgets behave like standard Tk widgets. To simplify this process, the Agentk animated widgets use the Tk requirement library for widget creation, querying, and modification created by Jeffrey Hobbs[1]. The widget package provides a framework for selecting components, creating subcommands and options, and integrating related procedures into a single namespace. Each of the animated widgets has as its sole component the canvas widget. This means that even though the animated widgets appear to behave like a standard label widget, they can make full use of the additional display capabilities of the canvas widget. Initial implementations used a label, but we found that the canvas was necessary to provide all of the desired behaviors.

The Agentk widgets are implemented in Tcl only, meaning that they can be used on any platform with no compilation. The entire package consists of about 4000 lines of code (plus an additional 10000 for the widget package), and individual widgets can be included or excluded as needed to further reduce the overhead. The widgets have been tested on various Unix platforms as well as Windows 95 and 98. All of the widgets can take advantage of the additional image formats and capabilities found in Jan Nijtman's Img extension[2] but it is not necessary to have the extension to use Agentk.

The remainder of this section focuses on two key issues addressed in the creation of

[1]See http://www.hobbs.wservice.com/tcl/script/widget/
[2]See http://purl.oclc.org/net/nijtmans/img.html

the animated widgets in Agentk. The first addresses performance across platforms, particularly important given that slower performance for animations results not simply in longer delays, but in a different appearance.

3.1 Maintaining platform independence

The loop that creates the animation effect (similar for each of the widgets) can be summarized as follows:

```
proc anim {} {
  # calculate next animation step
  ...

  after $delay anim
}
```

The `after` command waits for the period of time specified in the `delay` variable before executing the `anim` command.

Initial Agentk implementations ignored the time required to perform the calculations. On a slower or more heavily-loaded machine, the calculations may take significantly longer, resulting in an animation that is slower. Thus, a program may look very different depending on the platform. To address this problem, we calculated the time required to complete the calculations and subtracted it from the delay. If the resulting delay was less than zero, we adjusted the size of the steps taken by the algorithm. For example, a ticker might move two pixels instead of one, or a fading of text might make larger changes in color over a smaller number of steps.

In so doing, the animated widgets will look similar on any platform, regardless of the processor speed or machine load. The only perceivable difference is that the animation may be smoother on faster, less-heavily-loaded machines. By allowing for this automatic adjustment in performance, a programmer can write a script using animated widgets with confidence that it will appear the same to users on a variety of platforms.

3.2 Calculating an animation step

In fading between blocks of text or bitmaps (see the lower portion of Figure 4), the foreground color of the original information is gradually changed to match the background color, in essence fading the information away. At the same time, the new information (originally "invisible" because it starts as the background color) changes to match the foreground color. When the new information becomes closer (in an RGB-value sense) to the foreground color than the original information, it is raised to the top of the display stack.

When fading between two images (see Figures 1 and 4), it is impractical to change each pixel from the old color to the new – even a small 100x100-pixel image contains 10,000 pixels to repeatedly fade. Instead, the fade widget uses a stippling effect. Each image is divided into small squares, and the squares from the original are replaced with the squares from the new until the effect is complete. The user can specify the size of the squares or can specify the speed with which the animation will occur. If the size is specified, the speed will be dependent on the speed of the machine. A faster machine will be able to calculate and update the display more quickly than a slower one. If the speed is specified, the size of the squares will depend on the speed of the machine. Slower machines divide the images into larger squares, but the time required to fade from one image to another will be the same.

The tickering (and rolling) effects are accomplished by moving all of the items on the canvas horizontally (or vertically). As old textual and graphical items disappear from one side of the canvas, new items are added to the other. The number of pixels by which the display is shifted depends on the response time of the machine. Typically the display is shifted by a single pixel at each step, but slower machines may not be able to keep up with the desired speed at that rate. Our implementation regularly monitors the performance and increases the number of pixels if the system is being taxed. Note that this will result in an animation that is not as smooth, but it is more important that the appearance

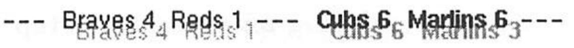


Figure 5: An example of automatic markups and history-based shadowing in a ticker widget showing sports scores. The bold text indicates a score that was recently updated, while the older score appears in plain text. The background shadow shows scores from ten minutes ago.

be similar on all machines.

4 Compensating for animation drawbacks

This paper has discussed ways that programmers can use our toolkit to incorporate animation into their programs in a platform-independent and resource-friendly manner. However, in creating a toolkit it is most important to consider the ability of the user to obtain the desired information. This section discusses ways that the Agentk animated widgets deal with three user-related concerns.

4.1 Highlighting changes with automatic markups

Although animation provides a means to smoothly show the current state of changing information, it can be difficult for a user to identify when and where a change has occurred. To address this problem, the Agentk animated widgets support automatic markups of text. These markups include boldface, italics, and coloring and can occur whenever the information in the widget is updated. For example, Figure 5 shows a widget that displays sports scores that are constantly downloaded from the Web. Recently changed scores are shown in bold text, while older scores appear in plain text.

A protocol is needed for removing the markups from older items. Agentk makes two options available to programmers: an automatic markup can be removed after a given period of time or after a given number of iterations of the display. In addition, the programmer can allow the user to reset the markups with a button press, mouse click, or other action. In our sports scores example, new scores could be highlighted for five minutes, or for ten iterations of the display. The programmer selects the option that seems best suited for the application.

Markups have long been established as a good way to highlight important parts of information. Systems like HtmlDiff [6] have used them to draw attention to changed information. The Agentk toolkit simplifies the integration of markups into situations where it is important to highlight changes. They are instantiated using command options that specify the desired style of markup and when they should be added and removed.

4.2 Showing previous states with history-based shadowing

Sometimes it is not sufficient to simply relate to a user that a change has occurred – it is necessary to communicate information about the nature of the change. To provide this capability, the widgets support *history-based shadowing*, a technique where a previous state of text is shown in the shadow of the current information state (see Figure 5). This technique captures the spirit of integrating supporting material described in [5] framed with a familiar shadow metaphor. In history-based shadowing, the shadow appears slightly offset horizontally and vertically from the original text and appears in a color that is between (in the RGB-value sense) the foreground and background colors. As such, it requires little extra space and is less obvious on first glance than the (more important) current state.

Similar to the automatic markups, the history-based shadowing for a given piece of information appears as soon as changes to the information are noted and can persist for a given length of time, a given number of iterations, or until the next change occurs. This gives the programmer the flexibility to choose a level of persistence that is best suited for the information.

History-based shadowing seems to be a good complement to animation and markups in maintaining awareness of information. The presence and persistence of the shadows is instantiated using Tcl-style command options. Shadowing provides a way to show not only the presence of changes to information, but also the state of the information prior to the changes.

5 Programming guidelines for animated widgets

One of the basic principles of interface design is to help the user maintain a sense of control over the actions on the screen. A user should not have to respond to actions but instead should control the rate at which information is assimilated. Yet animation seems to violate this principle: the flow of information typically is not under the direct control of the user. While our implementation attempts to empower the user by providing bindings for flow control, a programmer must be sensitive to the needs and desires of the user population. In fact, one criticism of animation in general is that it can be disruptive. The "blink" tag in HTML and animated advertisements on Web pages are often appropriately characterized as being annoying and disruptive. However, people have long used computer desktop accessories such as clocks and machine load monitors. What are the distinctions between these situations?

One distinction is that the burden of tolerating the constant changes is outweighed by the advantages these tools provide. Glancing at a desktop clock to obtain the time is far easier than running the date command or even looking at a wristwatch, and looking at a load monitor while running a compute-intensive program is far easier than running commands to determine the system load. A programmer should target application domains where the potential for knowledge is significant and the ability to start and stop the application is easy and apparent.

Another distinction is that the changes to the display are small, subtle, and predictable, allowing the user to adapt to the changing display to the point where it is barely noticeable. We hope that by providing smooth animations (many with user-controllable speeds), the applications will be less distracting. The programmer can assist in this by choosing appropriate sizes and speeds for the application. Users are more likely to use an application if it does not consume much desktop space and if it is not overly distracting.

Our experiences indicate that users are willing to use applications that employ animated widgets, though they generally do not use them continually. We regularly receive comments from users who started up an information awareness application for a few hours, whether it be to keep track of the score of a game in the heat of the playoffs or to watch for new news about the JFK Jr crash, but to our knowledge no one leaves the applications running continually. Programmers should not write applications that employ animation with the expectation that they will be used continually, but rather for short, well-defined periods of time, perhaps to monitor the traffic 5 and 6 pm every weekday, or to keep an eye on the scores of selected baseball games during the pennant drive and playoffs. If programmers do anticipate that it will be necessary to run the application at all times, alternate (non-animated) information delivery mechanims should be made available.

6 Related work

Animation has been used in various information visualization situations. Baecker and Small list several uses for animation, including among others identification, transition, demonstration, history, guidance, and feedback [1]. Some examples include the percent-done indicators for providing feedback [10] and animated icons for demonstrating use [2]. One use that has not been explored to the fullest is the application of animation to better utilize screen space. Cone trees [15] are an example of a visualization in which animation is used to show more information (in this case about hierarchies) than would otherwise be possible in a given space. We feel that this is a distinct advantage in information awareness applications.

Animation has been used to show dynamic information as well. Algorithm animation systems demonstrate how the dynamic processes of algorithms work [16]. The Tickertape interface scrolls text messages from email and other resources across the screen [14]. Bartram suggested the use of animation as an additional display dimension in the design of interfaces as well [3]. Yet, little work has been done on continuous, long-term animation for secondary awareness tasks. Animation has been dismissed as too intrusive, but here we argue that it can be used in some situations to maintain awareness without being too distracting.

Several other widget sets have been developed that allow the programmer to include animation in the interface. The Artkit toolkit allows programmers to create transition objects that describe how an object will move [7]. A reference to the transition object is then added to a graphical object to create the animation. Another toolkit, Amulet, was extended to include support for animation [11]. Animation can be attached to any value of any object, including position, color, or visibility. These and similar toolkits provide a great deal of power, yet they often can require significant effort by the programmer to achieve a desired result. The widgets in Agentk do not require the user to specify the details of the animation. In only a few lines of code, a programmer can specify variables, define animation behavior, and start a cyclic, repetitive animation that automatically updates when variables are changed by the program. This same behavior would be much more difficult with most other toolkits. At the same time, the Agentk animated widgets are structured such that new animated widgets can be added with significant code reuse.

7 Conclusions and future work

This paper has discussed the integration of animation into the Tcl/Tk toolkit. Incorporating animation and related techniques into a user interface toolkit makes it possible for programmers to include it in their applications using a familiar programming interface and makes it easier for users to keep a sense of control while maintaining a desired level of information awareness.

Our future work will concentrate on three areas. First, we plan to extend widget set to include other behaviors such as shrinking and growing, swiping, and others. Second, we are considering adding other optional techniques such as motion trails and slow-in/slow-out that may lessen the distraction caused by the animations. Third, we hope to make the execution more efficient, by making use of simplified calculation methods, improvements in machine speed calculations, and perhaps threading of the widget commands.

Programmer and user reaction to animated widgets has been encouraging, and we expect that the use will continue as the widget capabilities increase. The Agentk toolkit, including the animated widgets and most of the programs described in this document, is available at `http://www.cc.gatech.edu/~mccricks/agentk`.

Acknowledgments

The authors would like to thank the USENIX Association for providing several grants that made this work possible. We would also like to thank the GVU Center and the College of Computing at Georgia Tech for their support via grants, lab space, and machines. Finally, we would like to thank John Stasko and the Information Interfaces Group for their helpful comments on the Agentk toolkit and on this document.

References

[1] Ronald M. Baecker and Ian Small. Animation at the interface. In Brenda Laurel, editor, *The Art of Human-Computer Interface Design*, pages 251–267. Addison-Wesley, 1990.

[2] Ronald M. Baecker, Ian Small, and Richard Mander. Bringing icons to life. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 1–6, New Orleans, LA, May 1991.

[3] Lyn Bartram. Enhancing visualizations with motion. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis '98)*, pages 13–16, Raleigh, NC, 1998.

[4] Quasedra Y. Brown and D. Scott McCrickard. Cwic: Using images to passively browse the web. Technical Report 99-48, Georgia Tech GVU Center, Atlanta, GA, 1999.

[5] Bay-Wei Chang, Jock D. Mackinlay, Polle T. Zellweger, and Takeo Igarashi. A negotiation architecture for fluid documents. In *Proceedings of the ACM Symposium on User Interface Software and*

Technology (UIST '98), San Francisco, CA, November 1998.

[6] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. The AT&T internet difference engine: Tracking and viewing changes on the web. *World Wide Web*, 1(1):27–44, January 1998.

[7] Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '93)*, pages 57–67, Atlanta, GA, November 1993.

[8] Ioi K. Lam. Designing mega widgets in the Tix library. In *Proceedings of the 3rd USENIX Tcl/Tk Workshop*, Toronto, CA, July 1995.

[9] D. Scott McCrickard, John T. Stasko, and Q. Alex Zhao. Exploring animation as a presentation technique for dynamic information sources. Technical Report 99-47, Georgia Tech GVU Center, Atlanta, GA, 1999.

[10] Brad A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (SIGCHI '85)*, pages 11–17, April 1985.

[11] Brad A. Myers, Robert C. Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '96)*, Seattle, WA, November 1996.

[12] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[13] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

[14] Sara Parsowith, Geraldine Fitzpatrick, Bill Segall, and Simon Kaplan. Tickertape: Notification and communication in a single line. In *Asia Pacific Computer Human Interaction 1998 (APCHI '98)*, Kangawa, Japan, July 1998.

[15] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3d interactive animation. *Communications of the ACM*, 36(4):57–71, April 1993.

[16] John T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1:213–236, 1990.

Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience

Franc Brglez Hemang Lavana
Zhi Fu Debabrata Ghosh Lorie I. Moffitt
Steve Nelson J. Marshall Smith Jun Zhou

Dept. of Computer Science, Box 8206
NC State University, Raleigh, NC 27695, USA
http://www.cbl.ncsu.edu/~brglez/csc591b/

Abstract

This paper presents a class software project that was part of a recent experimental graduate course on Frontiers of Collaborative Computing on the Internet. *We chose Tcl/Tk to facilitate rapid prototyping, testing, and demonstrating all phases of the project. The major milestones achieved during this course are:*

- *rapid proficiency in Tcl/Tk that allowed each student to manipulate data and widgets, apply socket programming principles, and create a progression of client/server applications, from textbook cases to a unique client/server architecture prototype – driven by and matched to a well-defined collaborative project driver.*
- *universal server that supports any number of user-configurable clients, each accessible through a Web-browser on a Mac, Windows, or UNIX platforms. Prototype client configurations include: (1) collaborative document composition, (2) collaborative Tcl/Tk debugging and compilation, (3) collaborative design workflow.*

Keywords: client-server architectures, collaborative computing, Internet, Tcl/Tk, GUI.

1 Introduction

There is no consensus about the definition of *collaborative computing.* It appears that while *Computer-Supported Cooperative Work* (CSCW since 1960's) may have been eclipsed by *groupware*, the notion of collaborative computing is still being molded. Checking out a web-based search engine with keywords such as 'CSCW' returns about 44,470 links to related Web pages, 'groupware' returns about 228,940 links (mostly about Lotus Notes, NetWare, Office97, ...), while 'collaborative computing' returns only 8,857 links, several already listed under earlier keywords.

This paper is one of the two companion papers [1] that were initiated at the conclusion of the course on *Frontiers of Collaborative Computing on the Internet* (csc591-b, [2]). This course defines 'collaborative computing' as the hardware, software and structures that support a group of individuals working on related tasks – where hardware, software, data, *and* individuals may be distributed over a wide geographical area. 'Groupware' is a subset of collaborative computing: it is the software that allows communication, coordination and the sharing of information between distributed individuals and groups. Common examples of groupware are e-mail and video conferencing facilities, shared access to databases of documents and images, and applications such as shared white boards. A number of commercial systems are available to support such activities and the systems continue to evolve [3].

The main goal of csc591-b is to introduce collaborative computing as a distributed process, asynchronous and synchronous, that invokes, links, and executes

- data sets residing/generated on local/remote hosts;
- heterogeneous applications residing on local/remote hosts;
- floor control for collaborating teams distributed among local/remote hosts.

To make this process less abstract, we introduced the notion of a collaborative project:

An entity with distributed participants, distributed data sets and libraries, distributed tool sets and libraries, and objectives to be met by completing well-defined sequences of tasks – assigned by the the project leader, and subject to collaborative project activity and reviews.

* F. Brglez and H. Lavana have been supported by contracts from the DARPA/ARO (P-3316-EL/DAAH04-94-G-2080 and DAAG55-97-1-0345), and a grant from Semiconductor Research Corporation.

A project driver that matches this definition, and is within the scope of a single semester graduate class, is the creation of a distributed environment that supports collaborative document composition. The class instructor acts as a project leader and document editor, while students, having created a distributed environment first, maintain bibliographical databases, create graphics for embedded illustrations, write and compile the pre-assigned sections of the document. Most of such activity uses clients that interact with the local host of each participant, while periodic synchronization and sharing of data with other participants takes places when each client links to the common server.

While the environment as described above appears specific to document composition, the implementation, conceived as part of the class project, demonstrates a universal client/server architecture where the same server can now support any number of user-configurable, project-specific tasks or workflows. In order to complete most of the key objectives of the project before the end of the semester, we made Tcl/Tk [4] the scripting language of choice at the very beginning. The first few weeks of the course were devoted exclusively to the introduction and exercises in Tcl/Tk.

Our choice of Tcl/Tk for the class project in collaborative computing was also influenced by a number of favorable experiences with Tcl/Tk, each of which could also be shared and critiqued in the class environment, such as GroupKit [5], AgentTcl [6], user-configurable workflows [7], recording and playback toolkit [8], and toolkit for the web-browsers [9]. This paper introduces the *Asynchronous Group Server Architecture* (AGS) and a platform-independent client as it was conceived and implemented for a number of collaborative demos as part of the project in this course. The paper is organized into several sections as follows:

- Course Organization;
- Collaborative Project Driver;
- GUI and Client/Server Architecture;
- Server Design;
- Client Design;
- Collaborative Experiments;
- Conclusions.

A complementary architecture that also evolved from this class project, in particular the *Synchronizing Group Server Architecture* (SGS) and the transformation of single-user client-applications into collaborating client-applications is presented in a companion paper [1].

2 Course Organization

The class brought together a team of seven students, each with some programming experience under UNIX, WindowsNT, and MacOS. Few students had limited experience in Java programming, and only one student had significant experience in Tcl/Tk programming. The remaining students have learned about the rudiments of Tcl/Tk programming during the first few weeks of the course so that the remainder of the course could be devoted to a major joint class project that applied and extended the principles of client/server programming under Tcl/Tk. Specifically, the course was organized into three major sections:

- Rudiments of Tcl/Tk, including socket programming and client-server architectures;
- Case studies of various client-server architectures;
- Collaborative project outline, implementation hints, implementation reviews, final presentation and report.

The nominal textbook for the course was *Effective Tcl/Tk Programming* [10]. Chapters that were covered in some detail include Event Handling, Canvas Widget, Text Widget, Top-level Windows, and Interacting with Other Programs. Preceding the introduction of this textbook was a series of lectures on Tcl, mostly based on the material covered in more detail in [11, 12]. Tcl/Tk man pages and related links were made available on the course home-page [2].

Hands-on case studies, in class and as homework, of various client-server architectures covered *The Electric Secretary* in [10], *GroupKit5.1* in [5], *REUBEN* in [7]. In addition, the class was introduced to *WebWiseTclTk* in [9], which assisted in migrating the client application to the Web. Some of the homework assignments were completed and submitted as recorded sessions, using *RecordnPlay* in [8]. The introduction of AgentTcl in [6] was dropped due to lack of time.

A number of collaborative client/server architectures are known. Principally, they deal with single applications rather than workflows of applications, ranging from a shared calendar (The Electric Secretary) [10] to a shared whiteboard [5]; from collaborative visualization for health care [13] to collaborative editing of schematic diagrams [14]. This class project builds on the recent experiences with collaborative workflows of heterogeneous applications [7, ?, 8, 15] and the client-server architecture of *The Electric Secretary* in [10]. The latter served as the initial model for the *Asynchronous Group Server Ar-*

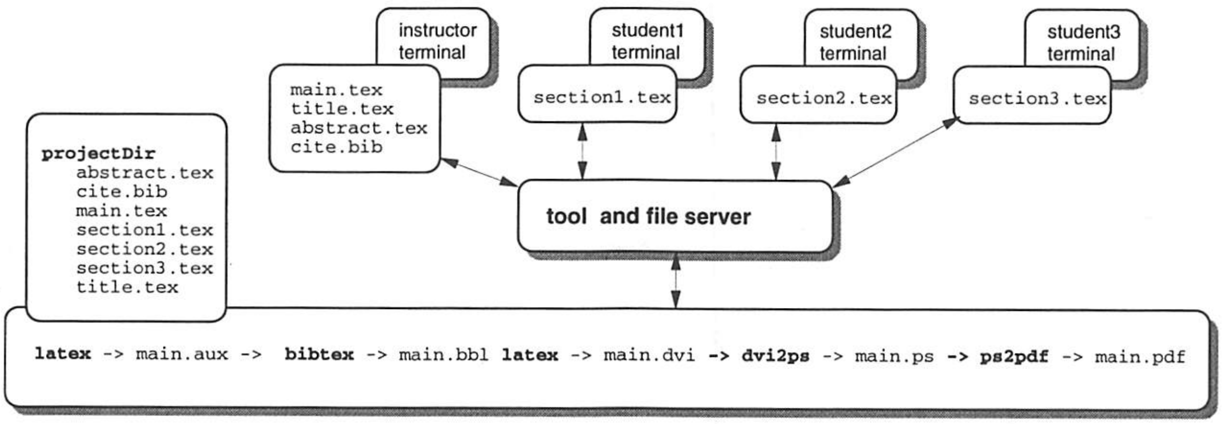


Fig. 1. An informal arrangement to collaboratively execute a document composition project.

chitecture (AGS) described in sections that follow.

3 A Collaborative Project Driver

The example that motivated and guided much of the class discussion and one that became the project driver, is first shown as an informal arrangement in Figure 1. The project involves the project leader (instructor) and a number of students. The objective of the project is to write a joint paper on the technology that will have enabled the class to devise and implement a client/server architecture to compile the paper in a collaborative mode. In the arrangement as shown in Figure 1, we assume

- all students have login access to a common tool and file server,
- each student will use a unique file name when writing a section assigned by the project leader,
- each student can create her own 'main' program that includes her section, and may be sections from others, to compile into a composite document that can be viewed or printed as a postscript or a pdf document by anybody in the class.

The compilation process itself may consist of all or some of the following steps:

1. first execution of `latex` [16] on file main.tex to output a file main.aux;
2. execution of `bibtex` on file main.aux to output a file main.bbl (to be cross-referenced with citations contained in a file *.bib);
3. second execution of `latex` on file main.bbl to output a file main.dvi (now containing the full document);
4. execution of `dvi2ps` on file main.dvi to output a file main.ps (ready for printing or viewing as a postscript file);
5. execution of `ps2pdf` on file main.pdf to output a file main.pdf (ready for printing or viewing as a pdf file).

This arrangement, while workable in principle, has a number of drawbacks:

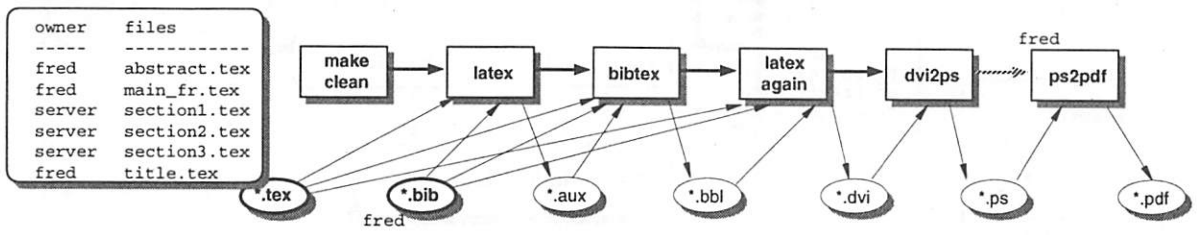
- it requires all participants to have a login account on the server;
- it is based on ad hoc coordination of files and symbolic links in the executable directory (owned by the project leader) versus the data files that are written by students in their home directories for which a symbolic link must be maintained in the executable directory by the project leader.
- it would require `ftp` in order for students to maintain their files on their local client hosts;
- it has no structure to render editing of files, or execution of the compilation sequence interactive and collaborative.

We next introduce an abstraction that formalizes the project description in Figure 1 such that we can also render it collaborative.

Formalizing the views. The document composition project illustrated in Figure 1 can be executed using a `makefile` – but only if data and tools reside on the same host. In general, this is not the case. The sequence of executable tasks in Figure 1 defines a simple workflow that can be represented with an acyclic Petri net graph consisting of data and tool nodes, with tool nodes acting as transitions that execute (or 'fire') only when all required input data is present at the inputs of the nodes. This is a special case of the more general case that involves graphs with cycles [7].

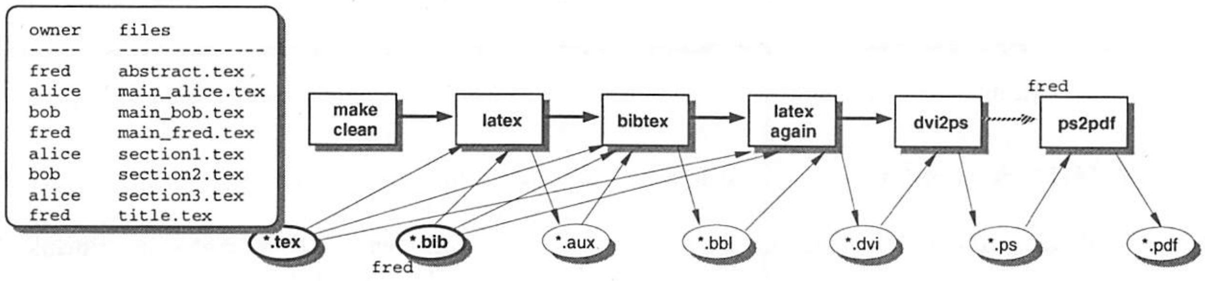
The graph representation, with distributed data nodes invoking tool nodes on any number of distributed file servers, is an effective GUI for the workflow client applications. In particular, such representation can readily support collaborative features

(a) Initial configuration of the project-specific workflow client (as initiated by project leader Fred)



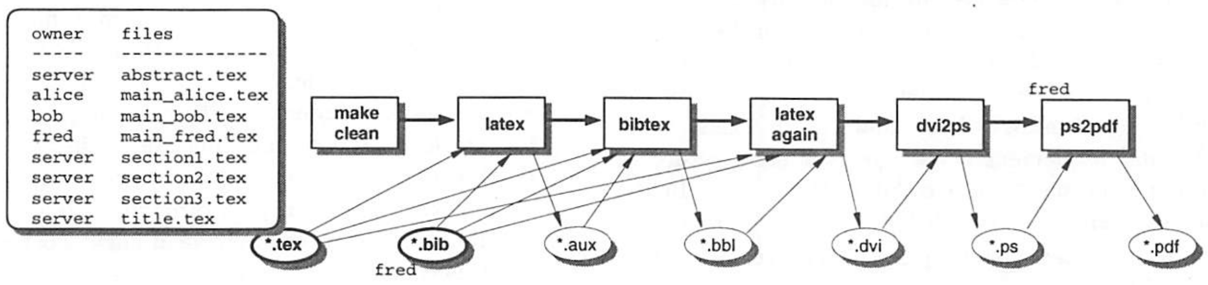
```
owner    files
-----    ------------
fred     abstract.tex
fred     main_fr.tex
server   section1.tex
server   section2.tex
server   section3.tex
fred     title.tex
```

(b) Asynchronous configuration of the project-specific workflow client (as continued by project team)



```
owner    files
-----    --------------
fred     abstract.tex
alice    main_alice.tex
bob      main_bob.tex
fred     main_fred.tex
alice    section1.tex
bob      section2.tex
alice    section3.tex
fred     title.tex
```

(c) Synchronizing configuration of the project-specific workflow client (as completed by project team and the leader)



```
owner    files
-----    --------------
server   abstract.tex
alice    main_alice.tex
bob      main_bob.tex
fred     main_fred.tex
server   section1.tex
server   section2.tex
server   section3.tex
server   title.tex
```

Fig. 2. Three views of a project-specific client workflow for collaborative edits and execution.

and user-configurability. Three views of the essential features of such a client interface are outlined in Figure 2: *set-up view*, *asynchronous view*, and *synchronizing view.*

Without loss of generality, we continue using the document composition project illustrated in Figure 1 as the illustrative example. To keep the presentation simple, the project leader (Fred) engages only two participants: Alice and Bob. The expected output from this project is a multi-section document on a specific topic, with Alice and Bob contributing technical sections, and Fred acting as the editor and also contributing the cover section, the abstract section, and the maintenance of the bibliography database.

`Set-up View.` Shown in Figure 2(a), this is a view of the client as initially configured by the project leader (Fred). In this project, task completion implies sequential execution of one or more tools from the set {*makeClean, latex, bibtex, dvi2ps, ps2pdf*}. With the exception of *makeClean*, these tools read instances of input files and write instances of output files. The task of the tool *makeClean* is to delete all of the intermediate file classes such as {**.aux, *.bbl, *.dvi, *.ps, *.pdf* } since any file instances in these classes can be regenerated by invoking the tasks that are driven by instances from *primary input* file classes such as {**.tex, *.bib* }. Generating and editing instances of files in these two classes represents the essential contribution to the project from each member of the team.

In the set-up view, Fred has the exclusive r/w ownership of all files in the class **.bib* as well as the files 'main_fred.tex', 'title.tex' and 'abstract.tex' in the in the class **.tex.* The files 'section1.tex', 'section2.tex', and 'section3.tex' are shown as owned by 'server', indicating that other team members are free to claim ownership to any of them. The file 'main_fred.tex' has a listing of any *.tex and *.bib files that are to be included in the compiled document. With the exception of *ps2pdf*, shown as controlled exclusively by Fred, there are no restrictions on execution of other tools. At this point, Fred can execute and test the task execution of the entire workflow by invoking *makeClean* and then *latex* on 'main_fred.tex', to be followed by other tool invocations until reaching *ps2pdf*. Alternatively, any segment of the tools could be chained for automated execution of the entire task sequence or any subsequence. At this stage, the files 'section1.tex', 'section2.tex', and 'section3.tex' are simple template files with tentative titles and are yet to be expanded and edited by Alice and Bob.

`Asynchronous View`. Shown in Figure 2(b), this view of the client depicts the work in progress, with contributions from Fred, Alice and Bob. Specifically, Alice claims ownership of 'section1.tex', and 'section3.tex', while Bob claims ownership of 'section2.tex'. In addition, each has created files 'main_alice.tex' and 'main_bob.tex', respectively, to execute any desired combination of the files from *.tex and *.bib class. In this view, the three participants work to a large extent independently, while each can always access, read, and process the files created by others. Typically, each may be editing the files on a local host rather than the server where all files are accessible to all the tools in the flow.

`Synchronizing View`. Shown in Figure 2(c), this view depicts the state of the client when the project leader Fred has scheduled a project review and a collaborative editing session in which all team members participate. While the team may reside at different locations, all communicate with each other, at the minimum via a chat-like window on the terminal screen. In the best case scenario, participants may communicate also via audio and/or a video channels. The main goals of the review are (1) to synchronize versions of input files generated by distributed team, (2) to review and edit individual files, and (3) to share the control of the flow execution for the final version of the document. This is an interactive and collaborative process in real time.

In the synchronizing view as shown in Figure 2(c), most files are returned to the ownership of the server – allowing any team member to access them in r/w mode for editing. In particular, we note that Bob has secured access to 'abstract.tex' – a file originally generated by Fred. The decision to access this file by Bob has been made after a brief discussion among all team members. In its crudest form, the mechanism by which others can observe Bob's editing is to download the revised file on prompts from Bob. In a more elaborate environment, others may observe Bob's editing in real time on their own terminal screen. Ultimately, one may expect an environment where two or more members may be editing the same file in real time in a user-friendly and an unambiguous manner.

Issues in rendering a client collaborative. Given the code for the stand-alone client application, the traditional approach is to re-write it as a client for collaborative application. This can be a formidable task, especially when all possible preferences for modes of collaboration cannot be anticipated in advance. Such a client may turn out to be user-unfriendly or confusing for a particular team. Simple preferences, such as whether and when should the scrollbars track for all participating collaborators, or should separate scrollbars be provided (and color-coded) for each participant, are at the core of such issues [5, 17, 18]. Such issues are addressed, and an effective solution proposed, in the companion paper [1].

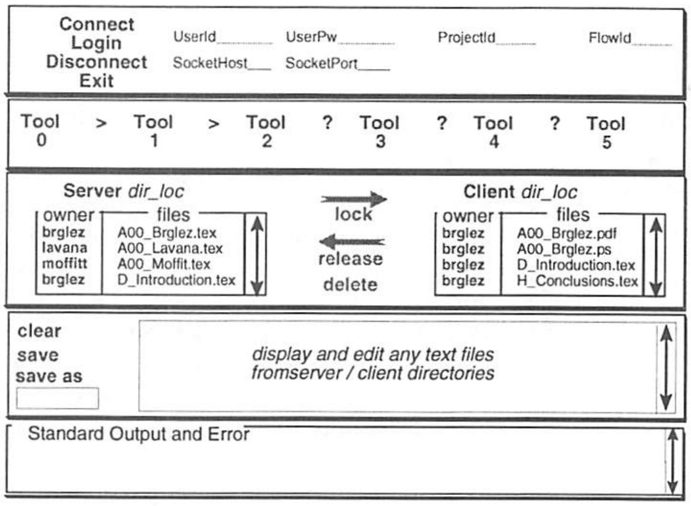
Generic examples of issues in rendering a stand-alone client collaborative, as introduced and discussed in the class setting, are included in the lecture notes [2] and the companion technical report [20]. The project driver example as introduced in Figure 1, and more formally in Figure 2, has been instrumental in arriving at the two-way partition of the server architecture to support two dynamic views of a collaborative project:

1. asynchronous view, defining parameters for an *asynchronous group server* (AGS);
2. synchronizing view, defining parameters for an *synchronizing group server* (SGS).

Due to limited time, the emphasis of the course was to prototype a client/server architecture that primarily supports the asynchronous view of collaboration and is described in the remainder of this paper. However, the simple demos, by the end of the class, that demonstrated the feasibility and the potential of the synchronizing view of collaboration, have also provided the direction beyond the class setting, leading to the companion paper on the SGS client/server architecture and implementation [1].

4 GUI and C/S Architecture

We have defined the concept of a collaborative project and the notion of a partitioning a set of project-specific tasks and data among project participants – giving rise to one or more workflow clients that are to be executed by participants in asynchronous and synchronizing modes. In addition, we identified behavioral classification of objects in



The workflow toolset is user-configurable and invoked by a specific *FlowId*, whereas *UserId* and *UserPw* are required to access a project-specific directory identified by *ProjectId.* Before clicking on the **Connect** button, user enters *SocketHost* and *SocketPort* . After clicking on the **Login** button, files in the project-specific directory are listed in the respective Server and Client widgets. Participants can release ownership of files by clicking on the files they own. File transfers between client and server directories can be executed with or without locking the ownership.

User-invoked tools are displayed as a tool chain and users can enable/disable any links by clicking on the > or ?. In this example, clicking on Tool0 invokes the tool, which on completion will also invoke Tool1, followed by Tool2.

Fig. 3. An example of a universal user-configurable workflow client interface.

the workflow client partitions, impacting the way we propose to implement a collaborative client. All of these factors have influenced the current view of the universal user-configurable workflow client interface and the corresponding server architecture.

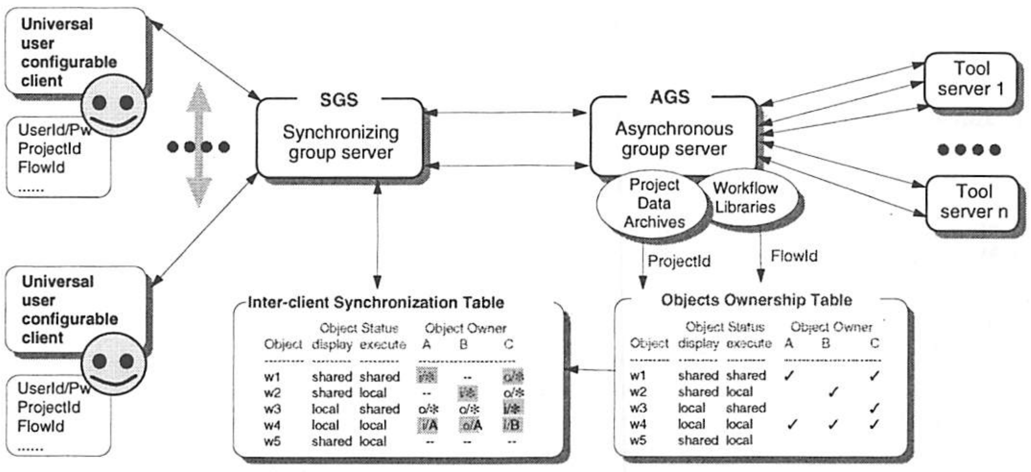
We first specify the GUI of the client, and describe how it relates to the proposed client/server architecture. This specification served as a blueprint for the student teams implementing several versions of both the server (in Tcl) and the client (in Tcl/Tk), as described in the subsequent sections.

GUI for the Client. The currently proposed and implemented workflow client interface consists of five major 'frames', each with a number of functions:

1. `loginFrame` supporting
   - buttons to *connect, login, disconnect, exit*
   - entries for *userId, userPw, projectId, flowId, socketHost, socketPort*
2. `toolFrame` supporting
   - buttons to invoke *tool0, tool1, tool2, . . .*. The bindings for the tools may be hard-wired in the initial versions of the client, but will be loaded later from a user defined configuration file, identified with a specific *flowID.*
   - connectors to connect/disconnect successive invocations of *tool0, tool1, tool2, . . .*
3. `filesFrame` supporting
   - server button that may toggle and display either server top-level project directory location or participant's subdirectory location on the server (the latter is created from project partitions by the project leader).
   - two-column listbox or textbox listing file owner and file name in the server directory or subdirectory.
   - client button that may toggle and display either local client project directory location or the interacting participant's local directory (subject to mutually agreed permissions).
   - two-column listbox or textbox listing file owner and file name in the respective client directory.
   - entries for buttons to *lock/release* ownership of selected files, button to *delete* a selected file, arrow buttons to *upload/download* a file to/from the server.
4. `editFrame` supporting
   - buttons to *clear, save, save as* files brought into the display and edit window.
   - text widget to display and edit any text file from the server or client directory (after clicking on the selected file). All files can be accessed for display, file owners can edit them.
5. `stdoutFrame` supporting
   - text widget to display any messages directed to standard output or to standard error.

A sketch of the proposed GUI for this client is shown in Figure 3, along with illustrative text of representative user interactions.

Client/Server Architecture. The proposed client/server architecture in Figure 4 matches the concept of a collaborative project as stated earlier: a number of workflows configured and partitioned by the project leader may be associated with each project; whereas participants may work in an asynchronous mode on the project partitions as well as



Inter-client Synchronization Table

	Object Status		Object Owner		
Object	display	execute	A	B	C
w1	shared	shared	i/*	--	o/*
w2	shared	local	--	i/*	o/*
w3	local	shared	o/*	o/*	i/*
w4	local	local	i/A	o/A	i/B
w5	shared	local	--	--	--

Objects Ownership Table

	Object Status		Object Owner		
Object	display	execute	A	B	C
w1	shared	shared	✓		✓
w2	shared	local		✓	
w3	local	shared			✓
w4	local	local	✓	✓	✓
w5	shared	local			

A number of workflows may be selected for each project, the objects in each workflow may be assigned a project and a team specific *object ownership table* by the project leader. This static table initializes the *Inter-client synchronization table* where activity permissions related to the objects can be changed dynamically by the participants, subject to the initialization constraints. For example, participant C can interact with object $w3$ and share the interactions with all participants $(i/*)$, while participants A and B can only choose to observe $(o/*)$ or not observe $(-)$ the same object. However, since all participants are initialized as owners of $w4$ (the shaded entries in this table correspond to the permission checkmarks in the initialization table), all can choose to interact in variety of ways with this object: A can interact with $w4$ alone, (i/A), with only participant B observing (o/A under the column B); C can interact with B only (i/B under the column C). Each participant can click on the entries in this table and 'toggle' the entry into the desired or allowable state. For example, the object $w3$ can be assigned to C as $i/*$, $o/*$, or $-$, while the same object can be assigned to B only as $o/*$ or $-$.

Fig. 4. The SGS/AGS client-server context and architectural features.

a synchronizing mode. This concept is reflected in the client/server architecture which itself is partitioned into an Asynchronous Group Server (AGS) and a Synchronous Group Server (SGS). For each project and workflow invoked by the participant, the AGS maintains not only the project data archives and workflow libraries but also an *Objects Ownership Table* preconfigured by the project leader. Once invoked, the Objects Ownership Table initializes the *Inter-client Synchronization Table* which interacts with SGS and the participants' clients.

A brief description of anticipated user-interactions in the interactive collaborative mode (where the Inter-client Synchronization Table can be changed dynamically by the project participants) is given in Figure 4. See the companion paper [1] for more details.

Project status by the end of semester. One student has implemented a version of the AGS server that supports all features as defined for the client in Figure 3 – except the file subdirectories for individual participants. A total of five similar but different clients have been independently developed by collaborating pairs of students. The *toolFrame* in one of these clients is user-configurable, so the client is truly universal.

In the document composition project, each participant can maintain/release ownership of files. A simple naming convention allows each member to execute the project partition independently of others, while the project leader can assemble and edit the complete document for immediate access and feedback to all. The client is invoked through a Web browser from any platform (UNIX, WindowsNT, MacOS) and collaborative execution can take place with distributed participants.

The paper concludes with highlights of AGS design, client design, and collaborative experiments conducted with the universal client/server architecture for three different applications: distributed document composition, distributed software debugging and compilation, and distributed experimental design environment.

5 Server Design

The design of a collaborative server was very critical to the success of this class project. There are several requirements of the server. It should support and maintain:

- connections from different students, after verification of their identity;
- several different projects and workflows accessible by the students;
- restriction for each project to access only those

specific tools that are required by the project;

- ownerships of data files in the project directory;
- data transfers between the server's project directory and the student's client host;
- invocation of tools in each project directory.

The first step was to define and develop a communication API (application programmer's interface) that can be used by the client/server to meet design requirements, as outlined above. We decided to employ *asynchronous communication* scheme, as described in [10]. The server was configured to understand a minimal set of commands. The mechanism for client/server communication is described next. The client sends a request to the server which consists of a string of the following form:

```
server_cmd arg1 arg2 ... argn client_cmd
```

The server processes the request from the client by invoking a procedure `server_cmd`, in a safe-interpreter, with n arguments. After the `server_cmd` completes its execution, the server parses the `client_cmd` string and replaces all occurrences of (1) `"%v"` with the result returned by `server_cmd` execution, and (2) `"%l"` with the length of the result returned by `server_cmd` execution. The new `client_cmd` string is then sent back to the client, which processes the string received in a safe-interpreter. However, whenever execution of `sever_cmd` results in an error, the server sends `error_result reason` to the client, instead of sending back the `client_cmd` string. The list of commands recognized by the server falls into four categories:

1. *Initial set-up/login process:* Once the clients establish a socket connection to the server, it is necessary to identify the user using the command:

```
login <user> <pswd> <project> <client_cmd>
```

If the user logs in successfully, then the following commands are made available to the client.

2. *Ownership of data:* The following commands allow the client to perform ownership related operations on a data file:

```
checkOwnership    <filename> <client_cmd>
grabOwnership     <filename> <client_cmd>
releaseOwnership <filename> <client_cmd>
```

A client can assume ownership of a data file, only if it is owned by the server. On releasing the ownership of a data file, the server becomes its owner. All three commands return the current ownership of the file.

3. *Data transfer:* Clients can get the list of data files, download, upload or delete a data file, as follows:

```
getList                          <client_cmd>
downloadFile <file>              <client_cmd>
uploadFile   <file> <data> <client_cmd>
deleteFile   <file>              <client_cmd>
```

4. *Tool invocation:* A client may invoke a tool on the server using the following command:

```
executeCommand <what args> <client_cmd>
```

This single command gives the server the flexibility to invoke any tool specified in `what args` without having to restart the server when a new tool need to be added to the services. However, it can also be major concern for security. We resolve this issue as described next.

Configuration Makefile. The server is designed to maintain several projects. Each project is restricted to access a limited set of tools only, depending on what tools are required by the project. A configuration `makefile`, stored in the project directory, determines the set of tools available for any given project. Each tool needs to be explicitly specified in the makefile along with the command line arguments necessary for its invocation. A typical entry in the makefile, which generates a pdf file from a postscript document, is as shown below:

```
document=A00_main
ps2pdf:
    ps2pdf $(document).ps $(document).pdf
```

This task may be invoked from the command line as `make ps2pdf` or `make ps2pdf document=A00_Brglez`. In the first case, the default value (A00_main) of the document is used, whereas in the second case, the document rootname is specified on the command line. Thus, a client would typically send the following command to the server to invoke the above task:

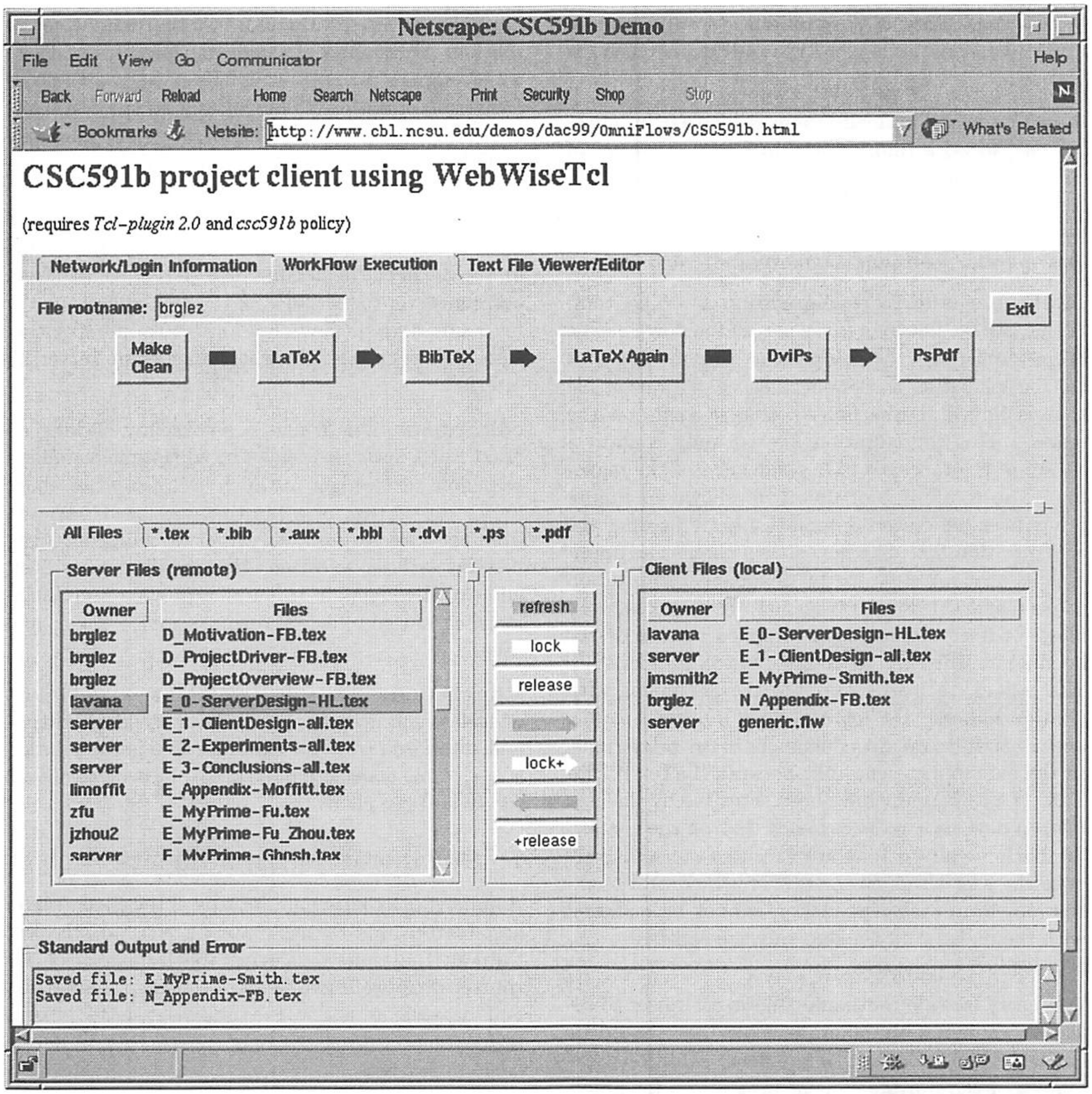
```
executeCommand "ps2pdf document=A00_Brglez" "puts {%v}"
```

The server will invoke this task, if available, in the project directory.

6 Client Design

As the course evolved, so did the versions of the client implementations and likewise, the versions of the server implementations – a true learning experience for everyone in the class. Unlike the server design which was completed by a single student who had prior experience with TclTk, the client design was completed by students who only learned TclTk during this course. While sharing the debugging experiences with each other, a total of five clients were designed independently by five student teams.

Each of the clients implemented the basic functionality as specified in in Figure 3 and the screenshot of



CSC591b project client using WebWiseTcl

(requires *Tcl-plugin 2.0* and *csc591b* policy)

Server Files (remote)

Owner	Files
brglez	D_Motivation-FB.tex
brglez	D_ProjectDriver-FB.tex
brglez	D_ProjectOverview-FB.tex
lavana	E_0-ServerDesign-HL.tex
server	E_1-ClientDesign-all.tex
server	E_2-Experiments-all.tex
server	E_3-Conclusions-all.tex
limoffit	E_Appendix-Moffitt.tex
zfu	E_MyPrime-Fu.tex
jzhou2	E_MyPrime-Fu_Zhou.tex
server	E_MyPrime-Ghosh.tex

Client Files (local)

Owner	Files
lavana	E_0-ServerDesign-HL.tex
server	E_1-ClientDesign-all.tex
jmsmith2	E_MyPrime-Smith.tex
brglez	N_Appendix-FB.tex
server	generic.flw

```
Saved file: E_MyPrime-Smith.tex
Saved file: N_Appendix-FB.tex
```

Fig. 5. Client design 1 (Here, the *toolFrame* is loaded from a configuration file).

each client, *demonstrated and tested for collaborative document composition functionality during the last session of the class*, is archived as part of the report posted on the class home page [2]. Each of the clients was tested through a Netscape browser and was executable from a UNIX, MacOS, and WindowsNT workstation, provided the browser has installed the WebWiseTcl [9], Safe-Tcl plugin, and the csc591b (course) policy. The latter is required to download and save the files from the server to the local client.

Each team was also responsible to contribute a subsection highlighting elements of TclTk used to achieve the required functionality. A representative GUI for the client is shown in Figure 5. Like all other clients designed in this course, this client is executable from window in a web-browser. This particular client is universal: the contents of the *toolFrame* are loaded from a user-specified configuration file, which is linked to the project-specific makefile maintained by the server. In addition, this client makes use of the B-widget Toolkit [19] to implement its GUI.

We use the the initial client interface specification in Figure 3 to briefly describe the GUI of the client in Figure 5.

loginFrame. The loginFrame is invoked by clicking on 'Network Information' in Figure 5. A set of entry boxes will prompt the user to enter 'UserId', 'UserPw', 'ProjectId', 'FlowId', etc. A set of well-placed buttons will allow user to 'connect', 'login', 'disconnect', 'loadFlow', 'removeFlow', and 'exit'.

toolFrame and reconfigurability. The toolFrame can be 'hard-wired' as part of the flow-specific client itself (e.g. the composition project flow), or can be loaded as a user-specified configuration file that is linked to the flow-specific *makefile* on the server. This configuration file can only contain a subset of the targets and dependencies of the makefile. The toolFrame of the client shown in Figure 5 has been generated automatically by loading the configuration file. The flow shown chains the applications that are used in the document composition project and can invoke 'Clean' to remove old work files, 'LaTex' to generate a compiled version of the document, 'BibTeX' to create citation indices, 'LaTeX Again' to load to update the document with citation indices, 'Dvi2ps' to create a postscript file, and 'Ps2pdf' to create a document in the pdf format. In the example shown, links between 'LaTex', 'BibTeX', 'LaTeX Again' have been activated by the user (shown now as arrows after each click on the connector bar), so all buttons are executed consecutively once the user clicks on 'LaTeX'. Such control of execution, from task i to task j, cannot be achieved with the make utility, where only the end-task j can be specified by the user.

The entry for 'file rootname' allows each user to select the name of the 'main.tex' file that should be invoked upon execution of the flow. Additional examples of the universal client invocation, for different sets of tasks, are shown in Figure 6.

filesFrame. The filesFrame has three parts: two listboxes that list files on the server and the client in the directories specified by the project name; a set of control buttons to allow a number of transactions take place between the two listboxes: file 'upload', 'download', 'release ownership', 'lock ownership', 'refresh', etc. These transactions take place once the filename has been highlighted in a specific listbox. By clicking on the file name, the contents of the file are displayed in the text window.

The display of the files in the listboxes can be filtered by clicking the appropriate class selection, e.g. *.bib would display files in this class only. In addition to files, the listboxes also show the current owner of the file. Only the file owner can modify or delete a file. There are two ways the owner can release ownership of a file: (1) by clicking on the file in the ownership field, or (2) by highlighting the file name and clicking the 'release' button. In either case the ownership field will change from 'userId' to 'server'. There are two ways the owner can lock ownership of a file when it is owned by 'server': (1) by clicking on the file in the ownership field (it will change from current server to 'userId'), or (2) by highlighting the file name and clicking the 'lock' button. In either case the ownership field will change from 'server' to 'userId'.

editFrame. This frame is invoked by clicking on 'Text File Viewer/Editor'. The editFrame includes a scrollable text widget in which user can write, modify, or delete text of a file that has been retrieved from the directory on the server or the client – depending on user selection. Besides the text widget, additional widgets in this frame provide functionality such as 'edit file', 'save on server', 'save', and 'clear'.

stdoutFrame. The stdoutFrame is a scrollable text widget which accepts inputs from standard output and standard error. Its main purpose is to maintain a log of all transactions taking place between the server and the client.

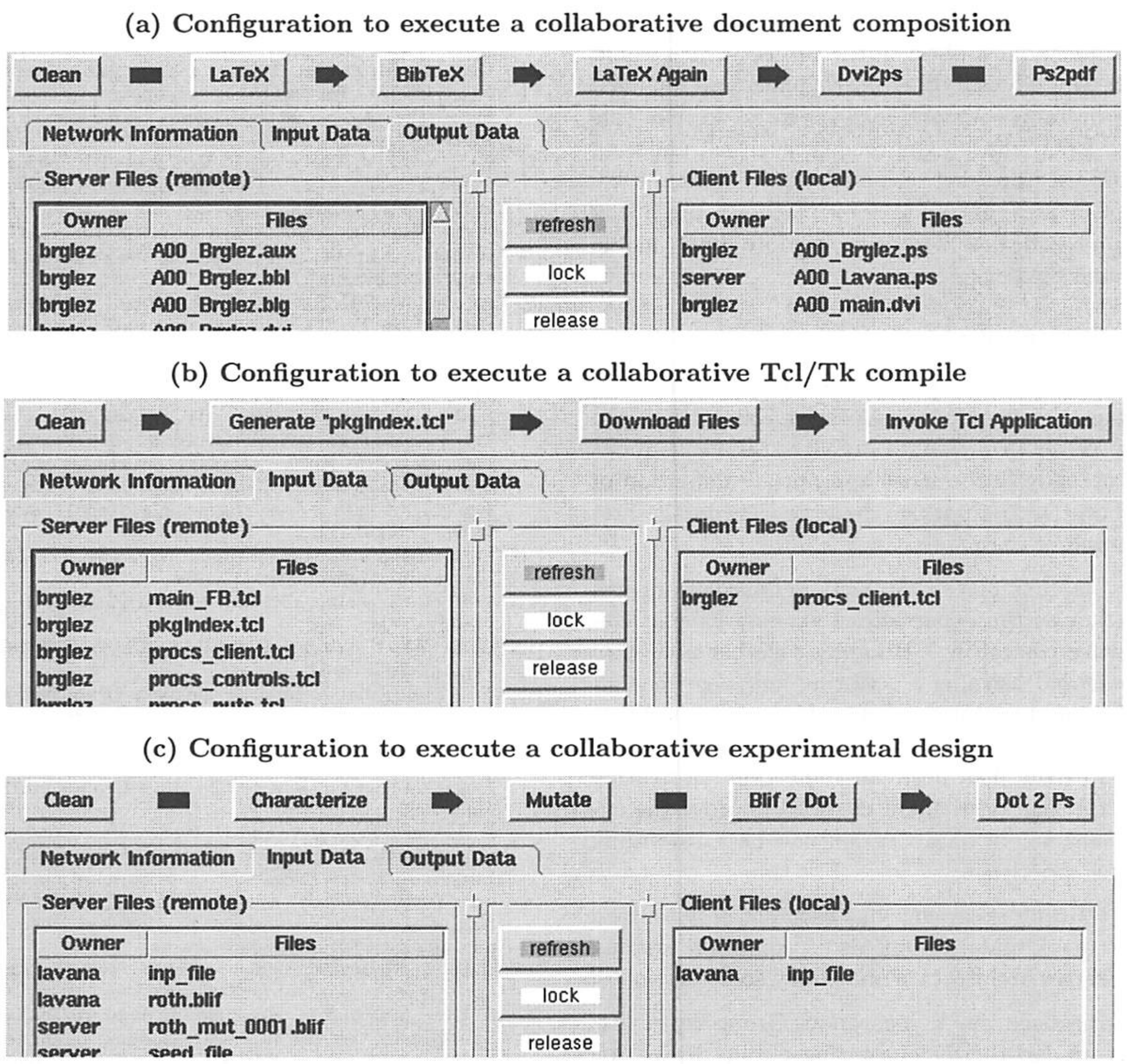
7 Collaborative Experiments

A number of experimental testing of the client/server architecture was taking place for most of the last third of the course. During the last week of the course, there were two sets of student presentations and demos: as a dry run and as a brief presentation/demo during the open house. These demos came in two sets:

Demo set 1. The complete set of demos that used the universal client described in Figure 5 is summarized in Figure 6:

(a) This flow executes the collaborative document composition as discussed in the preceding sections.

(b) This flow executes the collaborative Tcl/Tk compile. Here, there may be two or more participants working on a complex TclTk application. Each may be writing and testing a set of procedures in isolation. The question is: will they work together as expected. By uploading the files from several sources to the server, a combined version can be assembled and compiled on the server, then accesed for execution among the participants.



Owner	Files
brglez	A00_Brglez.aux
brglez	A00_Brglez.bbl
brglez	A00_Brglez.blg

Owner	Files
brglez	A00_Brglez.ps
server	A00_Lavana.ps
brglez	A00_main.dvi

Owner	Files
brglez	main_FB.tcl
brglez	pkgIndex.tcl
brglez	procs_client.tcl
brglez	procs_controls.tcl

Owner	Files
brglez	procs_client.tcl

Owner	Files
lavana	inp_file
lavana	roth.blif
server	roth_mut_0001.blif

Owner	Files
lavana	inp_file

Fig. 6. Three executable configurations of the universal workflow client.

(c) This flow executes the collaborative experimental design. A colleague may have uploaded a set of reference circuit files to the server to create a mutant class of circuits. These are characterized and laid out as schematics which can be accessed by another team for inspection and further analysis. Such flows are expected to play an important role in the collaborative design of experiments to test the performance of various algorithms [15].

Demo set 2. This demo set consists of the remaining four client implementations devised by student teams. Except for the universal user-reconfiguration of the task flow implemented in Figure 5, all of the client GUI designs have implemented the collaborative task flow for document composition in LaTeX as per original specification in the class. The minor difference in the interface are a reflection of individual preferences and the interpretation of the design specification. The screenshots of these client GUI designs are available in a technical report [20].

Notably, *all implementations* of the client have tested as executable through a web-browser. The latter included the implementation of the csc591b policy that allowed each participant to download and save files in the directory of the local host – thus overriding the nominal defaults of the web-browser. Currently, such fine-grained security policies are not as readily achievable with browser-based clients written in Java.

Final impressions. As the course concluded, there were really no surprises. The client/server architecture behaved as expected – it allowed multiple students to independently complete the writing assignments about a phase of the project which would then be shared with other participants and included in the overall document such an early draft of this paper. Preliminary experiments, such as rendering a single-user application collaborative under user-controlled preference also took place – a promising new approach, now described in more detail in the companion paper [1].

8 Conclusions

The material for this paper evolved as part of an experimental course on 'collaborative computing'. We are no closer to making a definite statement about what actually is the most appropriate definition of 'collaborative computing' than we were in the first paragraph of the paper.

In the context of this project, definitions are less important as the expectations we may have of collaborative computing. Bringing together a class of students and learning a scripting language that allows for rapid prototyping of user interfaces and networking concepts, and linking it all to a well-defined project driver, has been an important motivating factor for each participant eager to improve the environment where collaboration can be a rewarding learning experience.

The rewarding experience has been not only to learn the textbook material but also to question existing client/server architecture and to try some new ones.

The projects on the Asynchronous Group Server Architecture and the Synchronizing Group Server Architecture (SGS) along with the respective clients continues as a small project and both the server and the client software is expected to be released for use by the peer community during the year 2000. See

`http://www.cbl.ncsu.edu/software/`

for more details.

References

[1] F. Brglez and H. Lavana. CollabWiseTk: A Toolkit for Rendering Stand-alone Applications Collaborative. In *Seventh Annual Tcl/Tk Conference*. USENIX, February 2000. Also available at `http://www.cbl.ncsu.edu/-publications/#2000-TclTk-Lavana`.

[2] F. Brglez. Frontiers of Collaborative Computing on the Internet, A Graduate Course Experiment, January 1999. Two project reports, published after the completion of the course, are also available from the course home page under `http://www.cbl.ncsu.edu/~brglez/csc591b/`.

[3] Netmeeting Version 3.0. Published under URL `http://www.microsoft.com/netmeeting`, 1999.

[4] The Tcl/Tk Consortium. Published under URL `http://www.tclconsortium.org/`, 1998.

[5] GroupKit Version 5.1. Published under URL `http://www.cpsc.ucalgary.ca/grouplab/groupkit`, 1998.

[6] R. S. Gray. Agent Tcl: A transportable agent system, 1999. For up-to-date bibliography and software releases, see `http://agent.cs.dartmouth.edu/-software/agent2.0/`.

[7] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proceedings of the 34th Design Automation Conference*, pages 553–558, June 1997. Also available at `http://www.cbl.ncsu.edu/publications/-#1997-DAC-Lavana`.

[8] A. Khetawat, H. Lavana, and F. Brglez. Internet-based Desktops in Tcl/Tk: Collaborative and Recordable. In *Sixth Annual Tcl/Tk Conference*. USENIX, September 1998. Also available at `http://www.cbl.ncsu.edu/-publications/#1998-TclTk-Khetawat`.

[9] H. Lavana and F. Brglez. WebWiseTclTk: A Safe-Tcl/Tk-based Toolkit Enhanced for the World Wide Web. In *Sixth Annual Tcl/Tk Conference (Best Student Paper Award)*. USENIX, September 1998. Also available at `http://www.cbl.ncsu.edu/publications/-#1998-TclTk-Lavana`.

[10] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming*. Addison-Wesley, 1998.

[11] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] B. B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1997.

[13] TANGO: Collaboratory for the Web. Published under URL `http://trurl.npac.syr.edu/tango`, 1998.

[14] G. Konduri and A. Chandrakasan. A Framework for Collaborative and Distributed Web-Based Design. In *Proceedings of the 36th Design Automation Conference*, June 1999.

[15] H. Lavana, F. Brglez, and R. Reese. User-Configurable Experimental Design Flows on the Web: The ISCAS'99 Experiments. In *IEEE 1999 International Symposium on Circuits and Systems – ISCAS'99*, May 1999. A reprint also accessible from `http://www.cbl.ncsu.edu/-publications/#I999-ISCAS-Lavana`.

[16] (La)Tex Navigator, 1999. See `http://www.loria.fr/-services/tex/english/index.html`.

[17] S. Greenberg and M. Roseman. Groupware Toolkits for Synchronous Work. In M. Beaoudouin-Lafon, editor, *Computer-Supported Cooperative Work, Trends in Software Series*. John Wiley & Sons Ltd., 1998. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under http://www.cpsc.ucalgary.ca/projects/grouplab-/papers/1998/98-GroupwareToolkits.Wiley/Report96-589-09/report96-589-09.pdf.

[18] S. Greenberg. Real Time Distributed Collaboration. In P. Dasgupta and J. E. Urban, editor, *Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 1999. Also available as a Research Report 96/589/09, Dept. of Computer Science, University of Calgary, Calgary, Canada, under http://www.cpsc.ucalgary.ca/-projects/grouplab/papers/1998/98-Encyclopedia-Distrib/encyclopedia-realtime-collaboration.pdf.

[19] BWidget Toolkit, 1999. For more information, see `http://www.unifix-online.com/BWidget/`.

[20] F. Brglez, H. Lavana, Z. Fu, D. Ghosh, L. I. Moffitt, S. Nelson, J. M. Smith, and J. Zhou. Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience. Technical Report 1999-TR@CBL-01-Brglez, CBL, CS Dept., NCSU, Box 8206, Raleigh, NC 27695, May 1999.

Scripted Documents

Jean-Claude Wippler
Equi4 Software
jcw@equi4.com

ABSTRACT

Software used to be written as source code, which was then compiled and linked into a single machine-specific application program. With scripting languages, editable scripts are now executable without intermediate steps, but the dependency on lots of script files complicates robust deployment. A range of "wrapping" schemes are in use today to package scripts and extensions into a single file. The "Scripted Document" approach presented here goes further by offering a database-centric solution for packaging, installation, configuration, upgrades, as well as all application-specific data. An implementation for Tcl is described — using MetaKit as embedded database — with a summary of the experiences gathered so far.

Introduction

Scripting languages introduce the notion of quickly "gluing" together existing application and libraries, as well as custom-made extensions, in flexible ways. This paper takes a generalized look at the issues involved in deploying, maintaining, and evolving Tcl/Tk-based solutions. It shows how a more flexible solution, using an embedded database, can solve problems not addressed by current wrapping technologies.

One of the effects of moving away from monolithic compiled applications and towards scripted packages, is that the convenient single-executable packaging of the final "link step" is lost, and that many script files and extensions may need to be included in the installation process of the application. This is compounded by the fact that the script language implementation itself also needs to be installed, which also consists of many files and directories. As multiple scripted applications are installed, each with its own — often conflicting — requirements, as more platforms need to be supported, and when some of the files reside on shared file servers, installation all too often ends up becoming unwieldy.

The concept of "Scripted Documents" presented here offers a way to implement quick and conflict-free packaged deployment, while at the same time greatly simplifying upgrades and evolution of scripted applications. After a review of current approaches to packaging, the motivation and design of this new approach will be presented. Following are details of a Tcl/Tk implementation, examples of actual use, experience gained so far, and ideas for further refining and extending this concept. It will be shown that Scripted Documents are simple to implement, and solve a wide range of packaging problems encountered during and after deployment of scripted applications.

Why Package?

Packaged distribution is useful in a number of situations. It makes it possible to deliver software as a black box, to end users who want to use it without going into the underlying technology.

This is the case when a software product is sold as a user-installable system, or when it is intended to be used as a tool that should simply work "out of the box". In these situations, the software is expected to change very infrequently. The more convenient and robust such a packaging scheme is, the more likely it is to work without adding support overhead for the developer / vendor.

These end-user situations make it attractive to fully encapsulate the scripting language, in a way that it does not depend on, nor interfere with, anything else on the end-user's system. Robustness also means that the packaged application should not break when users install further applications later - even if some of these packages are not as careful to avoid interference with the rest of the system. This is a reason why including all shared libraries and script files inside the packaged application can be an advantage, despite the increased disk space requirements.

Note that in other situations, particularly when system integrators are available to maintain and customize a variety of software applications on a large system used by many users, packaged distribution may not be desirable, as it prevents the integrator from making changes they feel necessary in their environment. However, for the majority of cases, particularly with users running on Windows, Macintosh or personal Unix workstations, packaged distribution is the preferred solution.

Current Packaging Approaches

In this section we look at a number of current technologies used for packaging applications. Each of these must consider the following mix of components that comprise scripted applications:

1. A platform-specific *script language engine*, the "main interpreter", e.g. the Tcl binaries
2. Required platform-specific *language implementation libraries* (DLLs), e.g. libc.so
3. A standardized set of portable *library scripts*, the "runtime support", e.g. lib/init.tcl
4. A set of popular *core extensions*, either scripted or compiled, e.g. lib/Tk8.2/
5. Domain-specific *vendor-supplied compiled extensions*, e.g. lib/Mk4tcl/
6. *In-house extensions* to simplify common domain-specific tasks
7. Custom *compiled extensions* for performance and/or interfacing
8. The custom scripts representing the *main application* itself

Note that all but the last few items consists of generic code, which tends to be used in many different applications. It is code that tends to be stable, reliable and changes infrequently.

An important dimension is platform-independence. Scripts are portable: many files in the above list are identical for all platforms. When disk space was sparse, this used to be an argument to separate the scripts from the rest to allow sharing them (note that a fine-grained structure can also *complicate* maintenance).

There are several approaches today which simplify the process of packaging. All of them focus on initial deployment, not upgrading. Some act like an installer and unpack a large set of files on the target system; others "wrap" up everything into one file.

InstallShield / Wise

These represent the classical binary installers for Windows (RPM could be considered similar for Linux/Unix systems). They deliver their payload by creating a large number of files. This approach works well for a one-shot installation, and when subsystems are not shared. Once information is copied to central areas like the "c:\windows" directory and the registry, this approach can break down. It is not uncommon for Windows users to resort to a full machine re-installation after several months of active use.

tar x / configure / make install

This is the Unix-based source code install model (made popular by *GNU Autoconf*) for deploying applications on a variety of (mostly Unix-based) systems. It is *potentially* the most flexible approach, since it allows making changes at any level, but it has a flaw: it is oriented towards source-code deployment, which in turn depends on a *compiler installation* before it can be used. For scripted applications, installation of the proper compilation environment can become a major stumbling block. For end-users, this approach doesn't solve anything — it pushes the issues to another level.

Hybrid installation

Scripting languages like Tcl have long ago adopted a hybrid form, with the core implementation of the language being deployed through one of the two above means. They have been very successful by paying attention to detail and resolving issues to deal with many platforms and configurations. Tcl can be configured with static or shared libraries, on over two dozen different Unix systems, Windows, Macintosh, and more. Tk attempts to share as many configuration options with Tcl as it can (yet it cannot avoid introducing new ones, like where to find the X libraries on Unix). New releases and upgrades are simple to install — if you have the appropriate C compiler. This approach can still lead to conflicts when multiple versions of Tcl/Tk need to coexist on a machine.

All in all, the Tcl/Tk core is easy to deploy. The trouble starts with deployment of items 5-8 in the list given earlier. Extensions need to be told where Tcl/Tk has been installed, and they in turn need to be installed in a certain way for Tcl to find them. Given that these extensions usually come from different sources, the inevitable differences creep in. Getting a full set of packages, such as BLT, Expect, Itcl, Itk, and TclX installed is cumbersome. Upgrading one of the components to a new release can range from annoying to nearly impossible, if the extension does not properly address platform differences. Throw in a couple of useful (but not as widely ported) extensions, add support for both Windows and Unix, and all of a sudden you need to be an expert administrator. Not to mention the time it takes to install all this.

The following solutions are currently being used to package Tcl/Tk applications.

Stick with pure Tcl

This is a popular approach: install the Tcl/Tk core, and stay away from anything "non-standard". Write scripts, with perhaps a simple installation script to help users get started. This works well: no compiler dependencies, well-understood platform dependencies, and can be packaged as a few text files with a simple instruction on how to get started. Excellent examples are *TkCon* [8], and *BWidgets* [9].

Tcl2c

A refinement is to embed all scripts as C strings, and to rebuild Tcl/Tk as a *standalone shell*, which is easy to do with the *Plus Patches* [1]. This leads to a single executable combining all scripts, script libraries, and Tcl/Tk itself. This approach has several properties:

- For each platform to be deployed on, the packager needs a compiler and needs to repeat the entire wrapping process, even if the application itself consists only of scripts. A new "Wrap" mechanism is being developed, a refinement using a ZIP-compatible archive appended to the executable, to remove the need for re-compilation.
- It can lead to large executables, since every scripted application includes the complete Tcl/Tk core and all runtime scripts.
- It makes it possible to include binary extensions, by linking them in statically - though this again increases file size.

ProWrap and FreeWrap

ProWrap [3] (which is part of the TclPro product) and *FreeWrap* [4] collect all scripts and append them to a specially prepared standalone build of Tcl/Tk, i.e. at the end of the executable. On startup, the executable figures out how to get at those scripts and then reads and evaluates them as needed. According to ProWrap's web page documentation, it can pack and compress scripts as well as embed other files (but not shared libraries).

MkAppTcl

Richard Hipp's *MkAppTcl* [5] (and its predecessor *ET*) take a somewhat different approach, in that the perspective is shifted to having a main C/C++ application, which facilitates easily switching to Tcl/Tk requests. In a way, MkAppTcl maintains the model of *embedding* Tcl/Tk in C/C++, whereas the other approaches *extend* a scripted application with compiled extensions. The result is similar, with MkTclApp including a lot of support to ease the process of generating a standalone executable containing scripts, compiled extensions, and custom compiled code.

Application Logic vs. Application Data

Each of these approaches maintains a strict separation between *application logic* (scripted and compiled) and *application data*, which includes data files (textual or binary), database storage, and configuration files. In a way this is very useful, as the application is usually what the *developer* provides, and the data is what the *user* provides.

Yet this separation of code and data is at the heart of a range of some deployment and maintenance problems. New software releases no longer work with some documents, configuration parameters get detached from the application they apply to, documents can't be opened, applications don't know where their data is, and different application modules cause incompatibilities. These issues can be complex to deal with, requiring elaborate "environment settings" (Unix) and "registry settings" (Windows).

Let's face it: *real-world software deployment and maintenance is often a mess.* Installation introduces many problems, and leads to systems that are brittle.

A Different Perspective

As just described, wrapping consists of combining application logic (scripts) and application runtime support (language implementation) into a single executable, while application data remains separate.

Scripted documents choose a *different* separation, combining application logic and data into a single file, while separating out application runtime support.

The single application-specific portion combines all application logic plus data into a single database / file. It contains everything needed by the application, including all data that the application must manage after installation. This reflects a task-oriented and "document-centric" view, which is strengthened by the fact that scripted documents are *executable*.

Scripted documents represent *just* the application side of things. The scripting language implementation and all its support files are implemented as a separate and generic "runtime". The next section describes such a runtime for Tcl, called "TclKit".

Other systems exist where application logic is separated from a generic runtime, each as a single file. A popular example of these is in Java, where the runtime (i.e. the Java Virtual Machine) can read any number of "JAR" (Java Archive) files. Scripted documents differ from such solutions because the application specific piece contains both the application logic and modifiable application data.

This choice of separation into two components has several implications:

- Scripted documents are *portable,* allowing the vendor to deploy to users on any platform, simply by transferring the single file.
- Scripted documents store information *reliably*, when based on a transacted database package.
- Scripted documents have executable content, and look very much like a normal application to those who work with them. A scripted document represents the stored state of an application, as well as the active task while it is running.
- Scripted documents depend on *only* one other file, the runtime. This creates one new (very clear) dependency not present in wrapped applications, but removes the data file dependencies of wrapped applications.
- Runtimes are platform-specific, but have no application-specific content. There is exactly one runtime for each platform.
- A runtime requires no installation — it can reside in any directory on the execution search path.
- Un-installation is straightforward: remove the scripted document and the runtime.
- There is no need to install a scripting language, because the runtime is a self-contained system.
- Simple applications lead to tiny scripted documents. Scripted documents benefit from generic code shared through the use of a common runtime.

Effectively, *the scripting language core and its entire support system define an infrastructure, which scripted documents rely on in a well-defined way.*

Additional Uses for Scripted Documents

In the normal case, a scripted document will contain the logic for a single application, and all its data. However, it is possible to develop more powerful scripted documents, for use in a wider variety of situations. There are various ways to extend scripted documents:

- They can provide several user interfaces for the same application, for example a Tk based GUI version and a text based version, chosen at runtime depending on whether the user is able to run a GUI. The different versions of the interface manipulate the same data, stored within the scripted document.
- They can be used to package a number of related applications, with a simple file link or copy-and-rename causing them to behave differently. An example described later deploys a client/server system as a single file, making it easy to maintain consistent software.
- They can contain boilerplate utility code, activated from the command-line by specifying additional arguments. A generic startup script has been implemented for this, which also allows viewing and adjusting configuration parameters from the command line.
- They can take advantage of the transacted script storage to reliably update them selves through the network. Failure, or "rollback", will cause the system to revert to its prior state.
- They can be as general-purpose or as application-specific as you need them to be. Creating a scripted document that manages other scripted documents is one way to introduce configuration, customization, and/or upgrade management.
- Writing a scripted document which acts as design tool / IDE for itself or for other scripted documents would be a powerful way to simplify their development. Taken one step further, a large application could consist of one file which combines the final application (and therefore all its scripts), as well as an *embedded* development environment for it - a fascinating option for long-term maintenance. Scripted documents are the natural equivalent of "executables" in traditional compiled software.

Scripted documents retain the installation advantages of conventional wrapping technologies, but also solve the nightmares associated with configuration conflicts and upgrades, while greatly simplifying multi-platform deployment for the application vendor.

Upgrades and Evolution

When scripts are stored in a database, you can do things with them, which have traditionally been done only with application data. One of the most interesting is upgrading or evolving the application code itself.

Scripts can be altered while the application is running, and by the application itself. As a result, software upgrades cease to be a special issue; it's just a matter of storing new data over the old and committing the changes. This is just as safe as for changes to application data, when the underlying database uses a transaction/commit model to apply all changes.

Consistent and fail-safe updating is a critical feature for fundamental changes such as script upgrades. This mechanism is actually more robust that ordinary software development: if an installer, compilation, link step, or even a simple editor save over an existing script fails, there is a slight chance that the result will be corrupted or left in a inconsistent - and inoperable - state. This cannot happen with scripted documents, as implemented here. The worst that can happen is the introduction of a logic error: upgrading to a new release which does not work. From a technical perspective, there is nothing which can guard against such errors, but scripted documents do offer a way to minimize their impact: make a quick copy of the scripted document, which is a single file, to create a backup.

An interesting way to deploy and manage the application-logic of scripted documents is through upgrades over the net. The *TclDist* [12] example described later on is a generic (and portable) scripted document, which just asks for the URL of a web server to obtain the application, and which knows how to synchronize its scripts to that server. Such automatic upgrading has proven to be a popular feature for end users (e.g. as demonstrated by such technology in recent versions of Windows).

One of the reasons why scripted documents can be so flexible is because the underlying MetaKit [11] database used in this implementation supports *dynamic schema evolution* - adding and altering data structures of a scripted document is instant, and relies on the same transaction security as every other change. When an upgrade of the application *logic* is stored, the application can quickly extend or otherwise alter the application *data* stored in that scripted document, and thus support added functionality of any kind.

Updating the Runtime

The discussion so far has only covered changes to scripted documents. The assumption is that changes to the TclKit runtime are far more infrequent, since it contains only stable and generic code. If backward compatible, TclKit can be simply replaced as the need arises. Scripting languages in general have a track record of remaining extremely compatible over the years, even as very fundamental additions and improvements get added-in. Tcl is no exception.

If new releases of Tcl/Tk or MetaKit come out which are incompatible in a critical way, a new runtime will be created with a different name. This will lead to a new generation of scripted documents using that runtime (TclKit2?). As with all scripted documents today, there is no interference between such different versions. TclKit intentionally has no version number information in its name. It is intended to be a stable component in the world of scripted documents, and should go through extreme lengths to maintain backward compatibility. The primary way to achieve this goal, is to let TclKit err on the conservative side by lagging new Tcl/Tk and MetaKit releases (other than bug fixes).

Implementation Details

Implementing scripted documents in Tcl is relatively straightforward. Several issues need to be dealt with:

Database storage

Scripted documents store all scripts and data in a single file, are portable across platforms, and must be failsafe to protect a scripted document even after a system crash. The *MetaKit* database library meets these requirements. The fact that it is a good fit for scripted documents is not surprising, since it is the result of several years of development by the author — with many related design goals.

Executable content

Scripted documents are executable, by using platform-specific tricks to create the illusion that they are applications. In reality they are documents, which invoke the runtime as part of the startup process. On Unix, this is done in the same way as for shell scripts: making the first line "#!/bin/sh" and setting the execute-permission bit. On Windows, scripted documents must use a ".tkd" (TclKit Document) file extension or a small batch file. On Macintosh, the "file creator" must be set to a specific code.

Mixed script/data evaluation

The final step is to prefix all data stored as an embedded database with a special bootstrap header script to launch scripted documents correctly. For this to work, Tcl has been slightly modified to "source" a script(ed document) without getting confused if there is additional data tagged onto the end of the script. The first part of a file is scanned for a zero byte - if found, script reading stops there. If not, the file is assumed to be a normal script and is read in as usual.

Runtime package

The counterpart of a scripted document is its runtime, i.e. TclKit. Based on the *Plus Patch* [1] version of Tcl/Tk, a standalone executable has been created which consists of the Tcl and Tk core, all necessary supporting scripts, as well as the Tcl-aware version of MetaKit, called *Mk4tcl*. The *Trf* [6] extension is also included, to give access to *zlib* [7], which is very useful for packaging.

Let's examine in detail what happens when a scripted document is launched (using an imaginary "example.tkd" document on Windows as case study):

1. User double-clicks the "example.tkd" scripted document
2. Windows users should associate ".tkd" files to the tclkit.exe file, so that the runtime starts up upon double-click
3. TclKit opens the example.tkd file, reads everything up to the zero byte, and starts evaluating that data as a Tcl script.
4. Here is a basic version (without script compression or error handling) of the bootstrap header script:

   ```
   #!/bin/sh
   # \
   exec tclkit "$0" ${1+"$@"}
   package require Mk4tcl
   mk::file open doc $argv0 -nocommit
   eval [mk::get doc.scripts!0 text]
   return
   ```

   The first three lines are the standard Tcl'ish way of launching a script. These lines are only used under Unix, and skipped otherwise. The remaining lines initialize the MetaKit database in TclKit, reopen the scripted document as a database, fetch the "text" field of the first record in "scripts" as a string, and evaluate that string. If the script returns then break off (this is plus-patch specific; without it Tk enters an idle loop).
5. That's it, as far as the basic bootstrap into scripted document in Tcl is concerned! The first script stored in the database determines the rest.

Some comments:

- *Scripted documents rely on a very simple mechanism.* It's not much more than a single file / database storing one or more scripts, as well as any other type of data which needs to be stored. What makes them special is the way everything is packaged.
- The bootstrap process described so far is all there is to it. But the fact that all data now resides in a database, including the scripts needed to work with that data, and that this file is *modifiable by these same scripts* is what causes this to be an open-ended approach. You can make things as sophisticated as you like, by creating and including the appropriate scripts.
- All scripts in a scripted document can build on all of Tcl, all of Tk, and all of MetaKit - because all of these are always present. This means that as with any plain Tcl/Tk installation you have the power of scripting, networking, a cross-platform GUI, as well as a robust database, at your disposal. *That's a lot of infrastructure!*
- All changes to a scripted document are transacted, because this is how MetaKit works. Each scripted document can be used for large-scale and efficient demand-loaded data storage.
- On Unix, scripted documents can be used as command-line Tclsh-like applications, or as visual Wish-like applications. This is possible because the plus-patch version used in TclKit includes Tk as a run-time facility. A CGI application for web-server use need not activate Tk (and will not require X Windows) for example, while the same application can be also be used to run in fully GUI-oriented mode in other situations. Note that Windows cannot mix console and GUI modes, and that the Mac has no system console mode.

Startup Script

The first script in the database is what gets executed as the last step of the above bootstrap. Although one could store the main application script, normally a *command line interface* script (CLI) is stored there. This script examines command line arguments to offer a basic level of support for scripted documents. Among the functions it performs are:

- Adding or replacing database scripts using files or directories specified as argument
- Basic listing- and extraction facilities for scripts and other text-based information
- The ability to start up an alternate script by giving its name on the command line
- Defining a default startup script if no arguments are specified, or ignoring them
- Viewing and altering configuration options, which are also stored in the database
- A few utility procedures for scripts to easily access those configuration options

With no command line arguments, and if not specifically configured otherwise, the CLI simply looks for a script with the base name of the scripted document and executes it (if the scripted document is "example.tkd", the CLI would expect to find a database script entry called "example.tcl").

The CLI acts as an important safety net, in case a scripted document fails to start up properly (perhaps because a vital script was altered or deleted). As long as the first script is the CLI, one can examine the contents, restore scripts through the command line, launch scripts which validate or repair other parts of the database, and so on. In a way the CLI is like the *BIOS* of PC's, or the *bootstrap monitor* of embedded systems: rarely changed, but crucial during startup and recovery.

Note that scripted documents can only be "damaged" by scripts making improper changes and then actually *committing* those changes - system faults, crashes, even premature exits, will automatically revert the contents of a scripted document to the last committed state. Damage to scripted documents due to bypassing the database code is currently not detected, there are plans to implement checksums for some key data structures in MetaKit.

Using Extensions

With scripted documents, everything is fine... until you need to use dynamically loaded *compiled* extensions. These may be unavoidable to interface to existing code or to achieve acceptable performance for CPU-intensive tasks. Now, the ugly issue of platform dependence comes back — conflicting with the simple distinction used so far: a portable but *application-specific* scripted document and a platform-specific but *generic* runtime. The question is: where do you put these platform-specific extensions, in the context of scripted documents? There are a number of options:

Store the shared library on disk and remember path

This is the standard mechanism provided by Tcl. It can be streamlined by setting up *pkgIndex* and *auto_path* to automatically find and load extensions. The main problems of this approach are that different extension builds needs to be used on each platform, and that the cost of deployment and maintenance can be high.

Include the shared library in the scripted document

From a deployment standpoint, this is the preferred way. To be able to load such extensions, the package mechanism must be told how to extract the shared library to the disk, and then launch it. This solves the deployment and maintenance issue, but runs only on the platform corresponding to the library included in the scripted document.

Include several platform builds of the shared library

A refinement is to include several shared library builds, and to select the proper one for extraction at runtime. This extends the portability to all builds that have been prepared in this way, at the cost of increasing the size of the scripted document. Yet another refinement is to also include a Tcl-only version of the extension, which is used if no other build is suitable. The Tcl-only version could be a slower or more limited implementation, or a script that presents a clear explanation of why this functionality is not available.

Store shared libraries on an HTTP or FTP server

Given that the potential number of builds for compiled extensions can be very large, it is tempting to create a central place, where new builds get added over time. The package load mechanism can be similar to the above one, but instead of checking the scripted document, it would attempt to fetch the appropriate shared library from a remote server. This requires a trusted network environment.

Create "vendor-specific" scripted documents

For popular extensions, scripted documents could also be used to create a special "deployment package" for an extension such as Itcl/Itk/Iwidgets. It would have as only task to deploy those extensions, and to easily manage and upgrade them. Tcl scripts and other data are stored in this scripted document. In a way, this is an installer for that vendor's extension, which could take advantage of scripted documents to provide out-of-the box execution on any platform, demo's, documentation, an installation verifier, test suites, a cleanup facility, and perhaps net-based upgrades.

One last remark: TclKit supports "stubs", this is essential on some platforms to be able to load compiled extensions dynamically.

Practical Experience

The best way to summarize the experience with scripted documents so far, is: *they make deployment fun!*

CGI Applications

CGI applications are a great use for scripted documents. A typical Tcl-based CGI application consists of a variety of CGI scripts which must be installed. Of course, a Tcl interpreter needs to be installed. If it needs to store data in a database, a separate database extension must be compiled and installed. Finally, if the application contains pre-loaded data, all of those data files must be installed. That's a lot of separate pieces a webmaster needs to worry about to run your CGI application, particularly if they are not already using Tcl for their web site.

A simple example illustrates the scripted document solution. A CGI application implementing a simple bug-tracking system (written by Mark Roseman of TeamWave) started out having all the separate pieces identified above. It was turned into a scripted document with just a few simple changes:

- All scripts were added to a fresh scripted document with just the CLI code in it.

- The database access was altered to use the scripted document itself for storage, rather than an external MetaKit data file.
- A few configuration parameters were defined for easy customization.
- The system was extended to list some help text when not called as CGI process.

The whole process took perhaps half an hour. The result is a bug tracking system scripted in Tcl, consisting of just one file which runs on any system for which there is a corresponding TclKit executable (Windows, Macintosh, and several Unix'es). See the reference link to "BugCGI" [12] at the end to access / customize / use this system yourself.

WiKit

WiKit [12] is an implementation of a so-called "wiki-wiki web" — a tool which lets people enter and edit hyper-linked textual information over the net using nothing more than a web-browser. WiKit adds things like a search engine and a Tk user interface when used locally, and uses MetaKit to store all pages and the change history. WiKit includes about 2,500 lines of custom Tcl scripts. Several WiKit based systems have been in constant use since early 1999 (such as *the Tcl'ers Wiki* [14]); their biggest "drawback" being that it has become too easy to set up lots of them — not a good idea, in term of content management. Note how software deployment has become so simple and effortless, that it no longer matters...

TclHttpd

Matt Newman has created a scripted document containing Brent Welch's powerful tclhttpd server, thus creating what must be the most easy to deploy scripted HTTP server ever. Some additional scripting was introduced to make the scripted document behave as a normal file system for the server, and to allow merging contents from an external path with the files stored internally. There are a wide range of potential uses for this system, ranging from deploying a portable standalone documentation server to full HTTP-based client/server applications.

TclDist

As a first experiment in automatic script maintenance, a small generic bootstrap utility called *TclDist* [12] has been created as scripted document, which fetches a list of applications from the web. It then uses the *httpsync* [10] protocol to fetch the selected set of files that get inserted into the scripted document itself. The application that is created in this way is not only ready for use — it also knows how to check for new versions and how to update itself. This net-based mechanism turns out to be the easiest way by far to deploy scripted documents. It requires a trusted HTTP server and can perform efficient differential updates.

Large Scale Deployment

The first commercial project using scripted documents has been another excellent source of experience. For confidentiality reasons, this system will only be described here in general terms. It is a distributed application with two long-running processes used for unattended testing of specialized equipment, combined with two different user interfaces: a management interface used remotely by administrators to monitor correct operation and present reports and statistics, and a test-set interface to let a group of field engineers schedule ad-hoc and periodic tests to verify/stress certain parts of the system.

The whole application was built in pure Tcl/Tk plus MetaKit, and consists of two non-stop server processes (called STORE and CONTROL) and the two types of client applications, one of which is installed on about a hundred workstations (mostly Solaris and Windows NT). Due to the large number of machines on which the client packages needed to be present, and the fact that this project was at the start of a much more ambitious system with more modules and client application types to be added later, the project was an ideal context for deploying software as scripted documents.

The end result: a *single* scripted document of under 250 Kb, plus one TclKit runtime for each platform is all that is needed to deploy this elaborate client/server system. It consists of some 550 KB of Tcl scripts (roughly 14,000 lines of code, stored in compressed form), and it runs out of the box. The field engineers do not see Tcl, they simply see a single application, which they can store and use wherever they like.

The scripted document contains all application logic for all server and client processes, and adjusts its behavior based on the name under which it is stored. Copying the distribution file to a file called "STORE" sets up the database server. Copying it under the name "CONTROL" sets up the non-stop scheduler and equipment interface of this system. The same applies to both types of client applications in this project.

One further refinement makes this scheme complete: before the distribution is copied in this way, it is configured (through the standard scripted document command-line interface) with parameters which specify the central server host name and the port to use. As a result: all clients are pre-configured and know how to communicate with the main server. Since they are consistent, there is far less chance of mixing up incompatible versions of the different applications. Note also that all this "deployment" is portable, and that the only platform dependency is a check that the appropriate build of TclKit is present on the platform.

Upgrades and long-term evolution of this system is done by upgrading the STORE server, and then having all other processes synchronize their scripts to the server. This is automatically done whenever a client starts up, over the network, but it has so far only been carried out as an experiment. The goal is to make upgrades user-initiated but fully automatic, distributed by the STORE server, and to do this without bringing the system down, except when a few core components need to be changed. Further details still need to be worked out, but it looks like the current design will support this mechanism.

If you have ever deployed a system of such complexity, you will understand just how much administrative effort has been *avoided*, and why scripted documents are effective. Several features stand out in the above example:

Installation is a non-issue. Unless you call copying and renaming of one or two files "installation", it is clear that this aspect of deployment no longer matters.

Self-consistency. The combination of all application logic into a single file greatly reduces the chance of mixing up revisions and releases.

Net-borne upgrades. Whether on Intranet or Internet, the fact that scripted documents can safely update themselves from a central server makes upgrading easy, and can be automated as far as needed.

Data storage is implicit. The STORE server stores all application logic and data. Users of the system do not see this database as a separate entity in the system, it is simply "part of the server". The embedded database can be fully inspected and altered through the general-purpose Mk4tcl scripting interface.

Self-diagnosis. Since everything now happens inside scripted documents, a range of deployment-related tasks can be handled in a more generalized way, by including some diagnostic scripts as part of the scripted document. An example of this is a simple "dump.tcl" script, which has been written to inspect the contents of a scripted document — any scripted document.

Long term backups. A backup of a scripted document is more useful than a traditional database or file/tree backup, because it is more complete: it includes the essential data, but also the application that manages that data. Since they are also MetaKit data files, and since MetaKit has maintained backward file format compatibility since its first release in 1996 and will continue to do so, the data remains accessible. If needed, a utility script could be added to export all data in XML format. This export potential is permanently tied to the scripted document, even when transported to another system or stowed away for a long time.

Scripted documents fulfill the promise mentioned in the introduction: they solve "a wide range of problems encountered during and after deployment".

Future Work

The concept of scripted documents is simple. It took less than a week of work to make all the essential pieces play together. But what has been described so far only scratches the surface of how they could be put to use. Here are some ways in which scripted documents could be extended:

Generic self-development and self-management tools

This is the most obvious area of further development: tools to create, inspect, and alter scripted documents. As well as far more ambitious ones: an embedded IDE which lets you develop a scripted document with itself, a built-in visual design editor, generalized update-over-the-net tools, hooks for revision control, customer support embedded in an application, documentation viewers, an embedded debugger, and so on.

Getting rid of the compiler

The Achilles' heel of scripted documents is the shared libraries that need to be deployed as part of an application. A project by the author, called "Minotaur" [13], explores the usefulness of combining a high-performance portable Forth engine with scripting languages. One of the benefits is that tight loops and low-level functionality can then be written in Forth instead of C, maintaining the portability which makes scripting so attractive. Forth can also be used as glue to shared libraries at run time, removing one of the main reasons to create C interfaces.

It should be noted that scripted documents are not Tcl-specific, even though the current implementation is. There is no reason why the same concept could not be applied to Python, even shell scripts.

Conclusion

It takes more than just developing software to make software-based solutions work. This paper has presented a new concept for packaging and deploying scripting-based applications which changes the landscape of software installation, configuration management, but especially software upgrading and evolution.

The idea is to use a modular and component-based approach during development — an approach that matches the "gluing" nature of scripting well — and then to combine the pieces in easily deployed and out-of-the-box runnable "scripted documents" when delivering complete solutions. The redundancy of including some modules repeatedly in each software package is easily offset by the total clarity this mechanism introduces in the later stages of system evolution: deployment, maintenance, and upgrading.

The key difference between scripted documents and wrapping approaches are that scripted documents use a database to merge application logic and application data into a single file, while separating out the runtime. A sharp distinction is thereby made between the platform-specific *infrastructure* (the "runtime") and the platform-independent *application logic plus data* (the "scripted document").

Scripted documents solve the fragility problems inherent in separating application code and data. They greatly simplify multi-platform deployment for the application vendor, and introduce new opportunities for software upgrades.

This paper has shown how a simple mechanism can be implemented as a "TclKit" runtime for Tcl based on Tcl/Tk and the MetaKit embedded database, and how effective it ends up being in actual use. Refinements and future plans have been covered to highlight the current status and the potential of this approach.

Scripting languages are a major step forwards in programmer productivity because of their rapid application development style. Yet they fall short when it comes to the task of getting a software solution out into the hands of its users and walking away from it in a "problem solved, case closed" fashion. Scripted documents add that final step needed to make scripting not just effective to build with, but also very effective to deploy lasting turnkey solutions.

Acknowledgements

This paper originated from a suggestion by Matt Newman. It would not have been readable without the patience, feedback, and detailed input from especially Mark Roseman, as well as Matt Newman, Larry Virden, and Christian Tismer. I am grateful to each of them for their valuable advice and suggestions, and to Jan Nijtmans for extending his Plus patches to make Tcl work with scripted documents.

References

[1] The Plus patches, Tcl2c, and the new Wrap extension, *by Jan Nijtmans*
http://purl.oclc.org/net/nijtmans/plus.html

[2] Tcl Extension Architecture (TEA) *by Scriptics*
http://www.scriptics.com/products/tcltk/tea/

[3] ProWrap commercial wrapper, *by Scriptics*
http://www.scriptics.com/products/tclpro/wrapper.html

[4] FreeWrap, a Tcl/Tk standalone, *by Dennis LaBelle*
http://www.albany.net/~dlabelle/freewrap/freewrap.html

[5] MkTclApp embedding wrapper, *by Richard Hipp*
http://www.hwaci.com/

[6] Trf transformation framework, *by Andreas Kupries*
http://www.oche.de/~akupries/soft/trf/

[7] Zlib library, *by Jean-Loup Gailly and Mark Adler*
http://www.cdrom.com/pub/infozip/zlib/

[8] TkCon console interface for Tk, *by Jeffrey Hobbs*
http://www.purl.org/net/hobbs/tcl/script/tkcon/

[9] BWidgets user-interface widgets for Tk, *by Unifix*
http://www.unifix-online.com/BWidget/

[10] Httpsync protocol, *by Forrest J. Cavalier*
http://www.mibsoftware.com/httpsync/

[11] MetaKit database, *by Jean-Claude Wippler*
http://www.equi4.com/metakit/

[12] TclKit runtime and the sample BugCGI, TclDist, WiKit scripted documents *by Jean-Claude Wippler*
http://www.equi4.com/tclkit/

[13] Minotaur project, connecting Tcl, Python, and Perl, *by Jean-Claude Wippler*
http://mini.net/pub/ts2/minotaur.html

[14] Tcl'ers Wiki, a collaborative web site for the Tcl community, *maintained by Jean-Claude Wippler*
http://purl.org/thecliff/tcl/wiki/

The Tcl Extension Architecture

Brent Welch <welch@scriptics.com>
Michael Thomas <wart@scriptics.com>
Scriptics Corporation

Abstract

This paper describes goals and current state of the Tcl Extension Architecture (TEA). The goal of TEA is to create a standard for Tcl extensions that makes it easier to build, install, and share Tcl extensions. In its current form, TEA specifies a standard compilation environment for Tcl and its extensions. The standard uses `autoconf`, `configure` and `make` on UNIX and Windows. A longer term goal is to create an infrastructure that supports network distribution and installation of Tcl extensions. A standard build environment is a necessary first step to support automated compilation and distribution of extensions. This paper describes the current state of TEA, but we expect to continue to refine the standard and add to it as we gain experience with it.

Introduction

Compiling Tcl from the source distribution is easy. One of the strengths of Tcl is that it is quite portable and so it has been built on all kinds of systems including Unix, Windows, Macintosh, AS/400, IBM mainframes, and embedded systems. However, it can be a challenge to create a Tcl extension that has the same portability. The Tcl Extension Architecture (TEA) provides guidelines and samples to help extension authors create portable Tcl extensions. The TEA is a result of collaboration within the Tcl user community, and it will continue to evolve. TEA covers the following topics, which are described in more detail in the paper:

- Recommended Source Directory Structure:
- Standard Installation Directory Structure.
- Stubs Libraries.
- `Autoconf` and `Configure`.
- Standard `Make` Targets.
- A Sample TEA-Compliant Extension.
- Future Plans.

Standard Directory Structure

One goal of TEA is to make the process of configuring and building a Tcl extension very similar to building Tcl itself. In addition, building a Tcl extension depends on having access to the Tcl source distribution. You must configure and build Tcl before you build your extensions. The best way to organize your source code is to have Tcl and all your extensions under a common directory (e.g., `/usr/local/src` or `/home/welch/cvs`). This way the build process for an extension can automatically find the Tcl sources. The dependency on the Tcl source distribution is described later, and in the long term we hope to support building TEA-compliant extensions against a binary distribution of Tcl.

The Source Distribution

Table 1 describes the directory structure of the Tcl source distribution. The Tk distribution is similar. The directory structure divides the sources into generic and platform-specific directories.

Table 1 The Tcl source directory structure.

`tcl8.2`	The root of the Tcl sources. This contains a `README` and `license_terms` file, and several subdirectories.
`tcl8.2/compat`	This contains `.c` files that implement procedures that are otherwise broken in the standard C library on some platforms. They are only used if necessary.
`tcl8.2/doc`	This contains the reference documentation. Currently this is in *nroff* format suitable for use with the UNIX *man* program. The goal is to convert this to XML.
`tcl8.2/generic`	This contains the generic `.c` and `.h` source files that are shared among Unix, Windows, and Macintosh.
`tcl8.2/mac`	This contains the `.c` and `.h` source files that are specific to Macintosh. It also contains *Code Warrior* project files.
`tcl8.2/library`	This contains `init.tcl` and other Tcl files in the standard Tcl script library.
`tcl8.2/library/encoding`	This contains the Unicode conversion tables.
`tcl8.2/library/`*`package`*	There are several subdirectories (e.g., `http2.0`) that contain Tcl script packages.
`tcl8.2/test`	This contains the Tcl test suite. These are Tcl scripts that exercise the Tcl implementation.
`tcl8.2/tools`	This is a collection of scripts used to help build the Tcl distribution.
`tcl8.2/unix`	This contains the `.c` and `.h` source files that are specific to UNIX. This also contains the `configure` script and the `Makefile.in` template.
`tcl8.2/unix/dltest`	This contains test files for dynamic loading.
`tcl8.2/unix/`*`platform`*	These can be used to build Tcl for several different platforms. You create the *`platform`* directories yourself.
`tcl8.2/win`	This contains the `.c` and `.h` source files that are specific to Windows. This also contains the `configure` script and the `Makefile.in` template. This may contain a `makefile.vc` that is compatible with *nmake*.
`tcl8.2/win/`*`Build`*	*`Build`* is `Release` or `Debug`. This contains compiler output.

The Installation Directory Structure

When you install Tcl, the files end up in a different arrangement than the one in the source distribution. The standard installation directory is organized so it can be shared by computers with different machine types (e.g., Windows, Linux, and Solaris). The Tcl scripts, include files, and documentation are all in shared directories. The applications and programming libraries (i.e., DLLs) are in platform-specific directories. You can choose where these two groups of files are installed with the `--prefix` and `--exec-prefix` options to `configure`. The --prefix option specifies the root of the installation directory (e.g., /usr/local). The --exec-prefix option specifies a platform-specific directory (e.g., /usr/local/solaris-sparc) for applications and programming libraries. Table 2 shows the standard installation directory structure:

Table 2 The installation directory structure relative to the `--prefix` directory.

exec_prefix/bin	This contains platform-specific applications. On Windows, this also contains binary libraries (i.e., DLLs). Typical *exec_prefix* names end with `solaris-sparc`, `linux-ix86`, and `win-ix86`.
exec_prefix/lib	This contains platform-specific binary libraries on UNIX systems (e.g., `libtcl8.2.so`)
exec_prefix/lib/*package*	Contains `pkgIndex.tcl` files corresponding to binary libraries from *package* that are found in *exec_prefix*/lib.
bin	This contains platform-independent applications (e.g., Tcl script applications).
doc	This contains documentation.
include	This contains public .h files
lib	This contains subdirectories for platform-independent script packages. Packages stored here are found automatically by the Tcl auto loading mechanism.
lib/tcl8.2	This contains the contents of the `tcl8.2/library` source directory, including subdirectories.
lib/*package*	This contains Tcl scripts for *package* and its `pkgIndex.tcl` file. Example *package* directories include `tk8.2` and `itcl3.0.1`.
man	This contains reference documentation in UNIX *man* format.

The Package Mechanism

Extensions are installed and used as packages. A package can be one Tcl script, a collection of Tcl scripts, a binary library, or some combination of scripts and libraries. When you install an extension you need to update the package registry so that others can find the extension with `package require`. This section describes the default package management system, which uses a collection of pkgIndex.tcl files in directories along your auto_path.

The package registry is implemented by a collection of pkgIndex.tcl files. Tcl searches the directories listed in its auto_path variable for pkgIndex.tcl files. It also searches down one directory, so you can put your extensions and pkgIndex.tcl files into subdirectories of the main directories listed on auto_path. The default auto_path is

prefix/lib/tcl*version* *prefix*/lib *exec_prefix*/lib

Each pkgIndex.tcl file has one or more `package ifneeded` commands in it. These register Tcl commands that are called whenever a particular package is requested with `package require`. This section shows a few sample `package ifneeded` scripts to handle different configurations of packages.

Binary Library

A binary library (i.e., DLL) goes into the platform-specific lib directory. For example, you install your DLL into *exec_prefix*/lib/libfoobar1.0.so and you create a package index file in *exec_prefix*/lib/foobar1.0/pkgIndex.tcl, which contains this code:

```
package ifneeded foobar 1.0 [list load \
    [file join $dir .. \
libfoobar1.0[info sharedlibextension]]\
    Foobar]
```

Collecting all the binary libraries in one directory makes it easy to resolve dependencies among them and third-party libraries that support your Tcl extension. UNIX users may have to adjust their `LD_LIBRARY_PATH` to include the *exec_prefix*/lib directory. On Windows, these files are actually in the *exec_prefix*/bin directory, which is automatically searched. Keeping the pkgIndex.tcl files in separate directories keeps them independent.

Tcl Scripts

If your extension is just Tcl scripts, then it can be shared by users on different platforms. These libraries are typically kept in a subdirectory of *prefix*/lib, (e.g., /usr/local/lib/foobar1.0). You can use the `pkg_mkIndex` Tcl command to generate a pkgIndex.tcl file for your scripts:

```
pkg_mkIndex -verbose prefix/lib *.tcl
```

By default, pkg_mkIndex generates pkgIndex.tcl files that contain `tclPkgSetup` commands that use `source` or `load` indirectly. You might imagine that `package require` actually loads code, but by default is does not. Instead, the following `tclPkgSetup` command arranges for foobar.tcl to be sourced whenever the unknown command tries to find `Foobar_Init`, `Foobar_DoSomething`, or `Foobar_End`.

```
package ifneeded foobar 1.0 \
    [list tclPkgSetup $dir foobar 1.0 \
    {{foobar.tcl source {Foobar_Init
    Foobar_DoSomthing Foobar_End}}}]
```

The tclPkgSetup command is complex, so you should use the `pkg_mkIndex` command to generate these commands for you. If you use `pkg_mkindex -direct`, you can create a simpler package that is sourced immediately in response to the `package require` command. This direct package index looks like this:

```
package ifneeded foobar 1.0 \
    [list source [file join $dir
foobar.tcl]]
```

Library and Script Combination

If you have both scripts and binary libraries, then you can split your package into two parts: the shared part as Tcl scripts, and a platform-specific part as a binary library. The tricky part is building your pkgIndex.tcl file correctly. There are two problems. First, you can only have one `package ifneeded` command for a single package, so you need to specify something about the scripts and the library in one command. Next, you cannot predict the location of both parts of the package, so you have to assume they are installed in a standard location relative to the auto_path. Our preferred solution is modeled after the one in the SNACK sound extension by Kåre Sjölander.

Create a pkgIndex.tcl file in the *exec_prefix*/lib/foobar1.0 subdirectory. You will need to install a copy for each different platform that you compile for (e.g., solaris-sparc/lib/foobar1.0/pkgIndex.tcl and linux-ix86/lib/foobar1.0/pkgIndex.tcl.). This file loads the binary library and sources the Tcl script. We assume that the scripts are installed relative to the Tcl script library:

```
package ifneeded foobarArch 1.0 \
    "[list load \
    [file join $dir ../libfoobar1.0[info \
      sharedlibextension] Foobar] \;
    [list source [file join [file dirname \
      $tcl_library] \
        foobar1.0/foobar.tcl]]"
```

This example assumes the standard directory structure. It would be more general to search along the auto_path for the foobar1.0 subdirectory and then source its foobar.tcl file. If you have several script files, you can introduce a short procedure to source all of them. Or, you can have two pkgIndex.tcl files and require that your users require both (e.g., foobar and foobarArch). The packages can also require each other. For example:

```
package ifneeded foobarArch 1.0 \
    "[list load \
    [file join $dir ../libfoobar1.0[info \
      sharedlibextension] Foobar] \;
    [list package require foobar 1.0]"
```

Finally, by using the package unknown hook, you could define and use an alternate package manager. Newsgroup discussions have pointed out that searching for all the pkgIndex.tcl files can be slow on some systems. An alternate package manager could keep a more compact and efficient database, and perhaps have smarts about downloading packages from standard TEA repositories.

Autoconf, Configure and Make

In the past, UNIX, Windows, and Macintosh have different compilation environments. The advent of the free Cygwin tools have made it possible to standardize on `autoconf`, `configure` and `make` for the UNIX and Windows compilation environments. The Macintosh still uses Code Warrior project files, however. On Windows we use `make`, `sh`, and `autoconf` from Cygwin, and the `cl` (VC++) compiler from Microsoft.

The `autoconf` system is used to create Makefiles that have settings appropriate for the current operating system. By using `autoconf`, a developer on Windows or Linux can generate a `configure` script that is usable by other developers on Solaris, HP-UX, FreeBSD, AIX, or any system that is vaguely UNIX-like. The `configure` script, in turn, is used to generate the working Makefile. The three steps: setup, configuration and make, are illustrated by the build process for Tcl and Tk:

1. The developer of a source code package creates a `configure.in` template that expresses the system dependencies of the source code. They use the `autoconf` program to process this template into a `configure` script. The developer also creates a `Makefile.in` template. Creating these templates is described later. The Tcl and Tk source distributions already contain the `configure` script, which can be found in the `unix` and `win` subdirectories. However, if you get the Tcl sources from the network CVS repository, you must run `autoconf` yourself to generate the `configure` script.
2. A user of a source code package runs `configure` on the computer system they will use to compile the sources. The `configure` script examines the current system and makes various settings that are used during compilation.

Table 3 Standard `configure` flags.

`--prefix=`*`dir`*	This defines the root of the installation directory hierarchy. The default is `/usr/local`.
`--exec-prefix=`*`dir`*	This defines the root of the installation area for platform-specific files. This defaults to the `--prefix` value. An example setting is `/usr/local/solaris-sparc`.
`--enable-gcc`	Use the *gcc* compiler instead of the default system compiler.
`--disable-shared`	Disable generation of shared libraries and Tcl shells that dynamically link against them. Statically linked shells and static archives are built instead.
`--enable-symbols`	Compile with debugging symbols.
`--enable-threads`	Compile with thread support turned on.
`--with-tcl=`*`dir`*	This specifies the location of the build directory for Tcl.
`--with-tk=`*`dir`*	This specifies the location of the build directory for Tk.
`--with-tclinclude=`*`dir`*	This specifics the directory that contains `tcl.h`.
`--with-tcllib=dir`	This specifies the directory that contains the Tcl binary library (e.g., `libtclstubs.a`). (*Note*: this option is not yet supported.)
`--with-x-includes=dir`	This specifics the directory that contains `X11.h`.
`--with-x-libraries=dir`	This specifies the directory that contains the X11 binary library (e.g., `libX11.6.0.so`).

3. When you run `configure`, you make some basic choices about how you will compile Tcl, such as whether you will compile with debugging systems, or whether you will turn on threading support. You also define the Tcl installation directory with `configure`. This step converts `Makefile.in` to a `Makefile` suitable for the platform and configuration settings. .

4. Once `configure` is complete, you build your program with `make`. This steps checks your source files against the compiled files and reruns the compiler on any files that have changed since the last compilation. The results are binary libraries for extensions and executable programs for applications. `Make` is used for testing and installation, too. Table 5 on page 8 shows the standard `make` targets.

Standard `configure` Flags

Table 3 shows the standard options for Tcl `configure` scripts. These are implemented by a `configure` library file (`tcl.m4`) that you can use in your own `configure` scripts. The facilities provided by `tcl.m4` are described in more detail later. There are also many other command line options that come standard with `configure`. Some of these are meant to give you control over where the different parts of the installation go. However, because of the way Tcl automatically searches for scripts and binary libraries, you can mess up the Tcl installation by installing the libraries and the binaries in wildly different locations. Because of this, the Tcl installation procedures in the standard Makefile do not support the `--libdir` and `--bindir` options. In general, if the flags are not listed in Table 3, then they are not guaranteed to be supported by the standard Makefile template.

Examples

If you only have one platform, simply run `configure` in the `unix` (or `win`) directory:

```
% cd /usr/local/src/tcl8.2/unix
% ./configure flags
```

Use `./configure` to ensure you run the `configure` script from the current directory. If you build for multiple platforms, create subdirectories of `unix` and run `configure` from there. You are free to create the compilation directory anywhere (some prefer to keep all the generated files away from the sources.) Here we just use a subdirectory of the unix directory:

```
% cd /usr/local/src/tcl8.2/unix
% mkdir solaris
% cd solaris
% ../configure flags
```

Any flag with `disable` or `enable` in its name can be inverted. Table 3 lists the non-default setting, however, so you can just leave the flag out to turn it off. For example, when building Tcl on Solaris with the *gcc* compiler, shared libraries, debugging symbols, and threading support turned on, use this command:

```
configure --prefix=/home/welch/install \
   --exec-pre-
fix=/home/welch/install/solaris \
   --enable-gcc --enable-threads --
enable-symbols
```

Your builds will go the most smoothly if you organize all your sources under a common directory. In this case, you should be able to specify the same `configure` flags for Tcl and all the other extensions you will compile. In particular, you must use the same `--prefix` and `--exec-prefix` so everything gets installed together.

If you use alternate build directories, like the unix/solaris example above, you must specify `--with-tcl` when building your extensions. This is the directory where the Tcl build occurred. It contains libraries and a tclConfig.sh file that is used by the extensions configure process.

If your source tree is not adjacent to the Tcl source tree, then you must use `--with-tclinclude` or `--with-tcllib` so the header files and runtime library can be found during compilation. Typically this can happen if you build an extension under your home directory, but you are using a copy of Tcl that has been installed by your system administrator. The `--with-x-includes` and `--with-x-libraries` flags are similar options necessary when building Tk if your X11 installation is in a non-standard location.

Finding a working compiler

As the `configure` script executes, it prints out messages about the properties of the current platform. You can tell if you are in trouble if the output contains either of these messages:

```
checking for cross compiler ... yes
```

or

```
checking if compiler works ... no
```

Either of these means `configure` has failed to find a working compiler. In the first case, it assumes you are configuring on the target system but will cross-compile from a different system. `Configure` proceeds bravely ahead, but the resulting Makefile is useless. While cross-compiling is common on embedded processors, it is rarely necessary on UNIX and Windows. The cross-compiling message typically occurs when your UNIX environment isn't set up right to find the compiler.

On Windows there is a more explicit compiler check, and `configure` exits if it cannot find the compiler. Currently, the Windows configure macros knows only about the Visual C++ compiler. VC++ ships with a batch file, `vcvars32.bat`, that sets up the environment so you can run the compiler, `cl`, from the command line. You must run `vcvars32.bat` before running `configure`, or set up your environment so you do not have to remember to run the batch file all the time.

Installation Directories

The `--prefix` flag specifies the main installation directory (e.g., `/home/welch/install`). The directories listed in Table 2 are created under this directory. If you do not specify `--exec-prefix`, then the platform-specific binary files are mixed into the main `bin` and `lib` directories. For example, the `tclsh8.2` program and `libtcl8.2.so` shared library will be installed in:

```
/home/welch/install/bin/tclsh8.2
/home/welch/install/lib/libtclsh8.2.so
```

The script libraries and manual pages will be installed in:

```
/home/welch/install/lib/tcl8.2/
/home/welch/install/man/
```

If you want to have installations for several different platforms, then specify an `--exec-prefix` that is different for each platform. For example, if you use `--exec-prefix=/home/welch/install/solaris`, then the `tclsh8.2` program and `libtcl8.2.so` shared library will be installed in:

```
/home/welch/install/solaris/bin/tclsh8.2
/home/welch/install/solaris/lib/libtclsh8
.2.so
```

The script libraries and manual pages will remain where they are, so they are shared by all platforms. Note that Windows has a slightly different installation location for binary libraries (i.e., DLLs). They go into the *`exec_prefix`*`/bin` directory along with the main executable programs.:

Table 4 Standard autoconf macros defined by `tcl.m4`.

`SC_PATH_TCLCONFIG`	Locate the `tclConfig.sh` file and sanity check the compiler flags. This implements the `--with-tcl` option.
`SC_PATH_TKCONFIG`	Locate the `tkConfig.sh` file. This implements `--with-tk`.
`SC_LOAD_TCLCONFIG`	Load the tclConfig.sh file.
`SC_LOAD_TKCONFIG`	Load the tkConfig.sh file.
`SC_ENABLE_GCC`	Implements the --enable-gcc option.
`SC_ENABLE_SHARED`	Implements the --enable-shared option.
`SC_ENABLE_THREADS`	Implements the --enable-threads option.
`SC_ENABLE_SYMBOLS`	Implements the --enable-symbols option.
`SC_MAKE_LIB`	Generates definitions used to make shared or unshared libraries on various platforms.
`SC_LIB_SPEC`	Generates the name of a library and appropriate linker flags needed to link it.
`SC_PRIVATE_TCL_HEADERS`	Use this if you need to include `tclInt.h`
`SC_PUBLIC_TCL_HEADERS`	Locate the standard Tcl include files.

Using autoconf and the tcl.m4 File

Autoconf uses the m4 macro processor to translate the configure.in template into the configure script. Creating the configure.in template is simplified by a standard m4 macro library that is distributed with autoconf. In addition, a Tcl distribution contains a tcl.m4 file that has additional autoconf macros. Among other things, these macros support the standard configure flags described in Table 3.

The goal of tcl.m4 is to simply the configure.in templates used for extensions and to replace the use of the tclConfig.sh and tkConfig.sh files. The idea of tclConfig.sh was to capture some important results of Tcl's configure so they could be included in the configure scripts used by an extension. However, it is better to recompute these settings when configuring an extension because, for example, different compilers could be used to build Tcl and the extension. At present Tcl still generates tclConfig.sh, and some of the tcl.m4 macros depend on it. We plan to restructure the macros further so tclConfig.sh (and tkConfig.sh) will no longer be needed. So, instead of using SC_LOAD_TCLCONFIG, extensions will use a new macro that computes compiler settings.

Table 4 lists the public macros defined in the tcl.m4 file. The tcl.m4 file defines macros whose names begin with SC_ (for Scriptics). The four TCLCONFIG and TK_CONFIG macros listed in Table 4 will be eventually be replaced. There are other macros defined, but the following are the only ones guaranteed to persistStandard Make Targets
The sample Makefile includes several standard targets. Even if you decide not to use the sample Makefile.in template, you should still define the targets listed in Table 5 to ensure your extension is TEA compliant. Plans for automatic build environments depend on every extension implementing the standard make targets. The targets can be empty, but you should define them so that make will not complain if they are used.

Table 5 TEA standard Makefile targets.

all	Makes these targets in order: binaries, libraries, doc.
binaries	Makes executable programs and binary libraries (e.g., DLLs).
libraries	Makes platform-independent libraries.
doc	Generates documentation files.
install	Makes these targets in order: all, install-binaries, install-libraries, install-doc.
install-binaries	Makes binaries, and installs programs and binary libraries.
install-libraries	Makes libraries, and installs script libraries.
install-doc	Makes doc, and installs documentation files.
test	Runs the test suite for the package.
depend	Generates makefile dependency rules.
clean	Removes files built during the make process.
distclean	Makes clean, and removes files built during the configure process.

Using Stub Libraries

One problem with extensions is that they get compiled for a particular version of Tcl. As new Tcl releases occur, you find yourself having to recompile extensions. This was necessary for two reasons. First, the Tcl C library tended to changes its APIs from release to release. Changes in its symbol table tie a compiled extension to a specific version of the Tcl library. Another problem occurred if you compiled `tclsh` statically, and then tried to dynamically load a library. Some systems do not support back linking in this situation, so `tclsh` would crash. Paul Duffin created a *stub library* mechanism for Tcl that helps solve these problems.

The main idea is that Tcl creates two binary libraries: the main library (e.g., `libtcl8.2.so`) and a stub library (e.g., `libtclstub.a`). All the code is in the main library. The stub library contains a big jump table that has addresses of the functions in the main library. An extension calls Tcl through the jump table. The level of indirection makes the extension immune to changes in the Tcl library. It also handles the back linking problem. If this sounds expensive, it turns out to be equivalent to what the operating system does when you use shared libraries (i.e., dynamic link libraries). Tcl has just implemented dynamic linking in a portable, robust way.

To make your extension use stubs, you have to compile with the correct flags, and you have to add a new call to your extensions `Init` procedure (e.g., `Examplea_Init`). The `TCL_USE_STUBS` compile-time flag turns the Tcl C API calls into macros that use the stub table. The `Tcl_InitStubs` call ensures that the jump table is initialized, so you must call `Tcl_InitStubs` as the very first thing in your `Init` procedure. A typical call looks like this:

```
if (Tcl_InitStubs(interp, "8.1", 0) ==
NULL) {
    return TCL_ERROR;
}
```

`Tcl_InitStubs` is similar in spirit to `Tcl_PkgRequire` in that you request a minimum Tcl version number. Stubs have been supported since Tcl 8.1, and the Tcl C API will evolve in a backward-compatible way. Unless your extension uses new C APIs introduced in later versions, you should specify the lowest version possible so that it is compatible with more Tcl applications.

The Sample Extension

This section describes the sample extension that is distributed as part of TEA. The sample extension implements the Secure Hash Algorithm (SHA1). Steve Reid wrote the original SHA1 C code, and Dave Dykstra wrote the original Tcl interface to it. Michael Thomas created the standard `configure` and `Makefile` templates.

The goal of the sample extension is to provide a TEA-compliant example that is easy to read and modify for your own extension. Instead of using the original name, `sha1`, the example uses a more generic name, `exampleA`, in its files, libraries, and package names. When editing the sample templates for your own extension, you can simply replace occurrences of "exampleA" with the appropriate name for your extension. The sample files are well commented, so it is easy to see where you need to make the changes.

configure.in

The `configure.in` file is the template for the `configure` script. This file is very well commented. The places you need to change are marked with `__CHANGE__`. The first macro to change is:

```
AC_INIT(exampleA.h)
```

The `AC_INIT` macro lists a file that is part of the distribution. The name is relative to the `configure.in` file. Other possibilities include `../generic/tcl.h` or `src/mylib.h`, depending on where the `configure.in` file is relative to your sources. The `AC_INIT` macro is necessary to support building the package in different directories (e.g., either `tcl8.2/unix` or `tcl8.2/unix/solaris`).

The next thing in `configure.in` is a set of variable assignments that define the package's name and version number:

```
PACKAGE = exampleA
MAJOR_VERSION = 0
MINOR_VERSION = 2
PATCH_LEVEL =
```

The package name determines the file names used for the directory and the binary library file created by the Makefile. This name is also used in several `configure` and `Makefile` variables. You will need to change all references to "exampleA" to match the name you choose for your package.

The version and patch level support a three-level scheme, but you can leave the patch level empty for two-level versions like `0.2`. If you do specify a patch-level, you need to include a leading "." or "p" in it. These values are combined to create the version number like this:

```
VERSION =
${MAJOR_VERSION}.${MINOR_VERSION}${PATCH_
LEVEL}
```

Windows compilers create a special case for shared libraries (i.e., DLLs). When you compile the library itself, you need to declare its functions one way. When you compile code that uses the library, you need to declare its functions another way. This complicates the `exampleA.h` header file. Happily, the complexity is hidden inside some macros. The standard configure.in defines a `build_`*`Package`* variable with the following line, which you do not need to change:

```
AC_DEFINE_UNQUOTED(BUILD_${PACKAGE})
```

The `build_packageA` variable is only set when you are building the library itself, and it is only defined when compiling on Windows. We will show later how this is used in `exampleA.h` to control the definition of the `Examplea_Init` procedure.

The `configure.in` file has a bunch of magic to determine the name of the shared library file (e.g., `packageA02.dll`, `packageA.0.2.so`, `packageA.0.2.shlib`, etc.). All you need to do is change one macro to match your package name.

```
AC_SUBST(exampleA_LIB_FILE)
```

These should be the only places you need to edit when adapting the sample `configure.in` to your extension.

Makefile.in

The `Makefile.in` template is converted by the `configure` script into the `Makefile`. The sample `Makefile.in` is well commented so that it is easy to see where to make changes. There are a few variables with `exampleA` in their name. In particular, `exampleA_LIB_FILE` corresponds to a variable name in the `configure` script. You need to change both files consistently. Some of the lines you need to change are shown below:

```
exampleA_LIB_FILE = @exampleA_LIB_FILE@
lib_BINARIES = $(exampleA_LIB_FILE)
$(exampleA_LIB_FILE)_OBJECTS =
$(exampleA_OBJECTS)
```

You must define the set of source files and the corresponding object files that are part of the library. In the sample, `exampleA.c` implements the core of the Secure Hash Algorithm, and the `tclexampleA.c` file implements the Tcl command interface:

```
exampleA_SOURCES = exampleA.c tclexam-
pleA.c
SOURCES = $(exampleA_SOURCES)
```

The object file definitions use the `OBJEXT` variable that is `.o` for UNIX and `.obj` for Windows:

```
exampleA_OBJECTS = exampleA.${OBJEXT}
tclexampleA.${OBJEXT}
OBJECTS = $(exampleA_OBJECTS)
```

The header files that you want to have installed are assigned to the `GENERIC_HDRS` variable. The `srcdir` Make variable is defined during `configure` to be the name of the directory containing the file named in the `AC_INIT` macro:

```
GENERIC_HDRS = $(srcdir)/exampleA.h
```

Unfortunately, you must specify explicit rules for each C source file. The `VPATH` mechanism is not reliable enough to find the correct source files reliably. The `configure` script uses `AC_INIT` to locate source files, and you create rules that use the resulting `$(srcdir)` value. The rules look like this:

```
exampleA.$(OBJEXT) : $(srcdir)/exampleA.c
    $(COMPILE) -c '@CYGPATH@
$(srcdir)/exampleA.c' -o $@
```

The `cygpath` program converts file names to different formats required by different tools on Windows. On UNIX, the `CYGWIN` macro is simply defined to `echo`.

Standard Header Files

This section explains a technique you should use to get symbols defined properly in your binary library. The issue is raised by Windows compilers, which have a notion of explicitly importing and exporting symbols. When you build a library you export symbols. When you link against a library, you import symbols. The `BUILD_exampleA` variable is defined on Windows when you are building the library. This variable should be undefined on UNIX, which does not have this issue. Your header file uses this variable like this:

```
#ifdef BUILD_exampleA
#undef TCL_STORAGE_CLASS
#define TCL_STORAGE_CLASS DLLEXPORT
#endif
```

The `TCL_STORAGE_CLASS` variable is used in the definition of the `EXTERN` macro. You must use `EXTERN` before the prototype for any function you want to export from your library:

```
EXTERN int Examplea_Init
_ANSI_ARGS_((Tcl_Interp *Interp));
```

The `_ANSI_ARGS_` macro is used to guard against old C compilers that do not tolerate function prototypes.

Using the Sample Extension

You should be able to `configure`, compile and install the sample extension without modification. On my Solaris machine it creates a binary library named `exampleA0.2.so`, while on my Windows NT machine the library is named `exampleA02.dll`. The package name is `Tclsha1`, and it implements the `sha1` Tcl command. Ordinarily these names would be more consistent with the file names and package names in the template files. However, the names in the sample are designed to be easy to edit in the template. Assuming you use make install to copy the binary library into the standard location for your site, you can use the package from Tcl like this:

```
package require Tclsha1
sha1 -string "some string"
```

The `sha1` command returns a 128 bit encoded hash function of the input string. There are a number of options to `sha1` you can learn about by reading the man page that is part of the sample.

Future Directions

The short term goal of TEA is to provide a standard way to build Tcl extensions. We have created a sample extension for others to learn from, and we have been converting the widely used [incr Tcl], Expect, and TclX extensions to adhere to the standard.

The long term goal of TEA is to make distributing and installing Tcl extensions easy for the end user. We envision a system where open source extensions can be hosted in a common CVS repository, built automatically on a variety of platforms, and distributed to end users and installed automatically on their system. For this goal to succeed, we need to start with a standard framework for configuring and building extensions.

Web Links

The TEA home page is:

`http://www.scriptics.com/tea/`

The sample extension can be found at the Scriptics FTP site:

`ftp://ftp.scriptics.com/pub/tcl/examples/tea/`

The on-line CVS repository for Tcl software is explained here:

`http://www.scriptics.com/cvs/`

Acknowledgments

We would like to thank the following people and organizations: Paul Duffin, Jan Nijtmans, Jean-Claude Wippler, and Scott Stanton designed and implemented the stub library mechanism for Tcl. Steve Reid and Dave Dykstra wrote the Secure Hash Algorithm and the Tcl interface to it. Unified building procedures on all flavors of UNIX would not be possible without the autoconf tools from the GNU project, and the Cygwin tools from Cygwin extend this functionality to Windows.

A different version of this paper will appear as a chapter in Brent Welch's book, *Practical Programming in Tcl and Tk*, 3rd Edition, published by Prentice Hall.

XOTcl – an Object-Oriented Scripting Language

Gustaf Neumann
Department of Information Systems
Vienna University of Economics and BA
Austria
gustaf.neumann@uni-essen.de

Uwe Zdun
Specification of Software Systems
University of Essen
Germany
uwe.zdun@uni-essen.de

Abstract

This paper describes the object-oriented scripting language XOTcl (*Extended* OTcl), which is a value added replacement of OTcl. OTcl implements dynamic and introspective language support for object-orientation on top of Tcl. XOTcl includes the functionality of OTcl but focuses on the construction, management, and adaptation of complex systems.

In order to combine the benefits of scripting languages with advanced object-oriented techniques, we extended OTcl in various ways: We developed the filter as a powerful adapation technique and an intuitive means for the instantiation of large program structures. In order to enable objects to access several addition-classes we improved the flexibility of mixin methods by enhancing the object model with per-object mixins. We integrated the object system with the Tcl namespace concept to provide nested classes and dynamic object aggregations. Moreover, we introduced assertions and meta-data to improve reliability and self-documentation.

1 Introduction

XOTcl (pronounced exotickle) is an object-oriented scripting language providing several improvements targeted at the development and management of large systems. The base of our work was the OTcl, which is a Tcl extension introducing a dynamic object and class model by using solely the C-API of Tcl. XOTcl is a standard Tcl extension which can be dynamically loaded into every Tcl compliant environment (such as `tclsh`, `wish` or Wafe [23]).

A central property of scripting languages is the use of strings as the only representation of data. For that reason a scripting language offers a dynamic type system with automatic conversion. All integrated components (application specific extensions typically written in C) use the same string interface for argument passing. Therefore these components automatically fit together and can be reused in unpredicted situations without change. In [27] and [25] it is pointed out that *component frameworks* have proven to provide a high degree of code reuse, and are well suited for rapid application development. It is argued that application developers may concentrate primarily on the application task, rather than investing efforts in fitting components together. Therefore, in many applications scripting languages are very useful for a fast and high-quality development of software. Hatton [16] points out that the use of object-orientation in languages like C++ does not fit the human reasoning process very well. In [25] we argue that the identified deficiencies do not apply at the same degree on object-oriented scripting languages.

Tcl is equipped with functionalities like dynamic typing, dynamic extensibility and read/write introspection, that ease the glueing process of constructing systems from components. OTcl extends these important features of Tcl by offering object-orientation with encapsulation of data and operations, single and multiple inheritance, a three level class system based on meta-classes, and method chaining. Instead of a protection mechanism OTcl provides rich read/write introspection facilities, which allow one to change all relationships dynamically.

We continued and extended the design philosophy of Tcl and OTcl of providing freedom rather than constraints for the programmer. Examples are the support of dynamic changes and introspection mechanisms wherever possible. This design philosophy trades in expressiveness for protection in order to ease programming. However, a highly flexible language design implies less hard-wired protection against bad software architectures or bad programming style. We believe that no protection mechanisms can enforce the generation of coherent code/designs, so we focused on expressiveness by providing the programmer with more powerful constructs rather than on making decisions in her/his place.

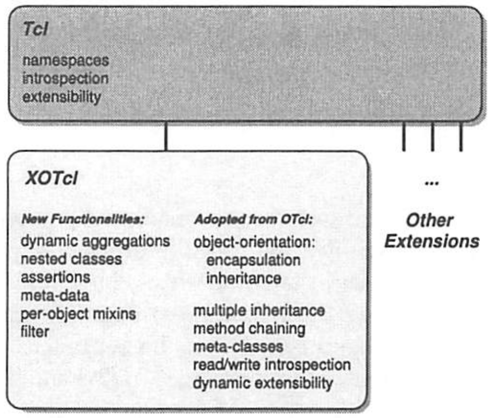


Figure 1: Language Extensions of XOTcl

The properties of OTcl described above provide a good basis for our work (see Figure 1). In the language design of XOTcl we focus on mechanisms to manage the complexity in large object-oriented systems, especially when these systems have to be adapted for certain purposes. Such situations occur frequently in the context of scripting languages. In particular we added the following support:

- *Filters* as a means of abstractions over method invocations to implement large structures, like design patterns, and to trace/adapt messages.
- *Per-object mixins*, as a means to give an object access to several different supplemental classes.
- *Dynamic Object Aggregations*, to provide dynamic aggregations through nested namespaces.
- *Nested Classes*, to reduce the interference of independently developed program structures.
- *Assertions*, to reduce the interface and the reliability problems caused by dynamic typing.
- *Meta-data*, to enhance self-documentation.

In this paper we describe theses functionalities from a language point of view. The implemented extensions provide additional functionality and lead to an improved performance in comparison to OTcl. However, we had to introduce a few incompatibilities to OTcl (discussed in Section 2 and in [33]). Since Wetherall and Lindblad provide in [32] a detailed presentation of OTcl and its design considerations, we focus here on the differences to XOTcl. The later sections introduce the new language constructs of XOTcl and discuss their usage. Finally we present a part of a larger application example (an XML-parser/-interpreter) based on design patterns, which is implemented using the new language constructs.

2 Language Constructs Derived from MIT Object Tcl (OTcl)

The `Object` command is used to create new objects. It provides access to the `Object` class which holds the common features of all objects. Objects are always instances of classes, but since objects from the most general class `Object` have no user-defined type, they may be referred to as *singular objects*. Every object can be dynamically refined with variables and with object-specific methods (using the `proc` instance method) at run-time. In the body of a proc, the predefined command `self` is used to determine the name of the current object. `self` can be used to obtain the following information about the current invocation:

- `self` (without parameters) returns the name of the currently executing object.
- `self class` returns the name of the class, which holds the currently executing method. Note, that it may differ from the object's class.
- `self proc` returns the name of the currently executing method.

A reader with OTcl knowledge will note, that there is a difference to the realization of these object informations to OTcl. XOTcl uses commands to obtain this information, whereas OTcl uses three implicit variables for this purpose (`self`, `class`, and `proc`). This change makes the internal calling conventions of XOTcl methods compatible with Tcl procedures. This has the advantage that the methods are accessible in XOTcl via namespace-paths (see Section 5). For compatibility XOTcl provides the compilation option `AUTOVARS` to set these variables automatically (with a slight performance disadvantage).

Every object is associated with a class over the `class` relationship. Classes are special objects with the purpose of managing other objects. "Managing" means that a class controls the creation and destruction of its instances and that it contains a repository of methods ("instprocs") accessible for the instances.

The instance methods common to all objects are defined in the root class `Object` (predefined or user-defined). Since a class is a special (managing) kind of object it is managed itself by a special class called "meta-class" (which manages itself). One interesting aspect of meta-classes is that by providing a constructor pre-configured classes can be created. New user-defined meta-classes can be derived from the predefined meta-class `Class` in order to restrict or enhance the abilities of the classes that they manage. Therefore meta-classes can be used to instantiate large program structures, like some design patterns. The meta-class may hold the generic parts of the structures. Since a meta-class is an entity of the programming language, it is possible to collect these in (customizable) pattern libraries for later reuse (see Section 7 for example or [25] for more details).

XOTcl supports single and multiple inheritance. Classes are ordered by the relationship `superclass` in a directed acyclic graph. The root of the class hierarchy is the class `Object`. A single object can be instantiated directly from this class. An inherent problem of multiple inheritance is the problem of name resolution, e.g. when two superclasses contain a method with the same name. XOTcl provides an intuitive and unambiguous approach for name resolution by defining the precedence order along a "*next-path*" for linearization of class and mixin hierarchies (see [32, 24] for details), which is modeled after CLOS [4]). A method can invoke explicitly the shadowed methods by the predefined command `next`. It mixes the next shadowed method on the next-path into the execution of the current method.

The usage of `next` in XOTcl is different to OTcl: In OTcl it is always necessary to provide the full argument list for every invocation explicitly. In XOTcl, a call of `next` without arguments can be used to call the shadowed methods with the same arguments (which is the most common case). When arguments should be changed for the shadowed methods, they must be provided explicitly. In the rare case that the shadowed method should receive no argument, the flag `--noArgs` must be used.

An important feature of all XOTcl objects is the read/write introspection. The reading introspection abilities are packed compactly into the `info` instance method which is available for objects and classes. All obtained information can be changed at run-time with immediate effect dynamically. Unlike languages with a static class concept, XOTcl supports dynamic class/superclass relationships. At any time the class graph may be changed entirely using the `superclass` method, or an object may change its class through the `class` method. This feature can be used for an implementation of a life-cycle or other intrinsic changes [20] of object properties (in contrast to extrinsic properties e.g. modeled through roles [14, 20] and implemented through per-object mixins [24]) . These changes can be achieved without loosing the object's identity, its inner state and its per-object behavior (procs and per-object mixins).

3 Filters

The filter is a novel approach to manage large program structures. It is a very general interception mechanism which can be used in various application areas. We have studied the use of filters for design patters in detail (see [25] and Section 7). Other useful application areas are monitoring of running systems (tracing and debugging), adaptation at runtime, implementation of proxy services, etc.

Definition 1 *A filter is a special instance method registered for a class C. Whenever an object of class C receives a message, the registered filter is invoked instead of the object's methods. The filter may handle this message and/or can decide to forward it to the object's methods.*

Usage of Filters In order to define a filter two steps are necessary: an filter-`instproc` has to be defined and the filter has to be registered using the `filter` instance method. This registration tells XOTcl, which instprocs are filters on which classes. Every filter consists of three (optional) parts:

```
ClassName instproc FilterName args {
  pre-part
  next
  post-part
}
```

The distinction into three parts is just a naming convention for explanation purposes. The pre- and post-part may be filled with any XOTcl-statements. In general the filter is free in what it does with the message. In particular it can (a) pass the message through (using the `next`-primitive), it can (b) redirect it to another destination, or it can (c) decide to handle the message itself (see Figure 2).

When a filter instproc is executed at first the instructions in the *pre-part* are processed. Then the filter might call the actual method through `next`. The filter can take the result of the actual method (returned by the next-call) and can modify it. After the execution of "next" the *post-part* is executed. Finally the caller receives the result of the filter instead of the result of the called method.

As an example we define a class `Room`. Every time an arbitrary action occurs on a room instance, the graphical sub-system should change the display of that particular room. A filter can handle the necessary notifications (here only output messages):

```
Class Room
Room instproc observationFilter args {
  puts "room action begins"
  set result [next]
  puts "room action ends -- Result: $result"
  return $result
}
Room filter observationFilter
```

When the filter is registered (last line) every action performed on an instance of `Room` is noticed with a pre- and a post-message to the standard output stream. We return the result of the actual called method, since we don't want to change the program behavior at all. When for example an instance variable is set on the instance of `Room r1`:

```
r1 set name "room 1"
```

the output is:

```
room action begins
room action ends -- Result: room 1
```

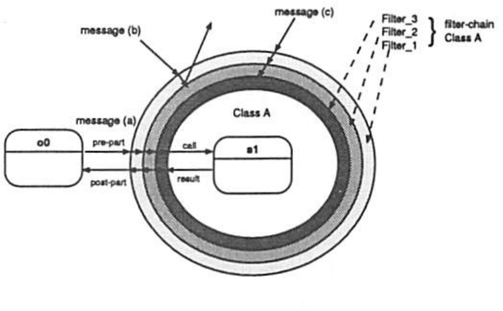


Figure 2: Cascaded Message Filtering

Filter Chains Each class may have a chain of filters which are cascaded through `next` (see Figure 2). The `next` method is responsible for the forwarding of messages to the remaining filters in the chain one by one (in registration order) until all pre-parts are executed. Afterwards the actual method is invoked and finally the post-parts are processed. If a next-call is omitted the filter chain ends in this filter method. In the following example two filters are registered, one for observation purposes and one for counting calls.

```
Room set callCounter 0  ;# set class variable
Room instproc counterFilter args {
  incr [self class]::callCounter
  next
}
Room filter {counterFilter observationFilter}
```

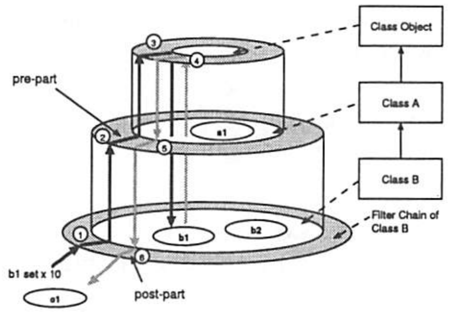


Figure 3: Filter Inheritance

Filter chains can also be combined through (multiple) inheritance using the `next` method. Filter chains of the superclasses are invoked using the same precedence order as for inheritance (see Figure 3). Without sophisticated efforts a powerful tracing facility can be implemented. E.g. a filter solely for offices distinguishes `Office`-rooms from other rooms:

```
Class Office -superclass Room
Office instproc officeFilter args {
  puts "actions in an office"
  next
}
Office filter officeFilter
```

A simple call to an instance `o1` of class `Office`, like:

```
o1 set name "office 1"
```

increments the counter and produces the output:

```
actions in an office
room action begins
room action ends -- Result: office 1
```

Introspection of Filters Filters are ordinary instprocs and have therefore access to all XOTcl functionalities including the full introspection facilities. Furthermore, filters require per-call information for reasoning or delegation purposes, i.e. information about the caller's and the callee's environment and the invocation record is required. The following options are additionally available:

- *objName* `info calledproc`: Returns the originally invoked proc.
- *objName* `info calledclass`: Returns the (presumably) called class.
- *objName* `info callingclass`: Returns the class from which the filtered call was invoked.
- *objName* `info callingproc`: Returns the proc from which the filtered call was invoked.
- *objName* `info callingobject`: Returns the object from which the filtered call was invoked.
- *objName* `info regclass`: Returns the class on which the filter is registered.
- *ClassName* `info filters`: Returns the list of filters registered for a class.

These methods return empty strings, when the desired information does not exist. The options with the prefix `calling` represent the values of `self`, `self proc`, and `self class` in the invoking method.

Tracing This example primarily demonstrates the inheritance of filter chains. Since all classes inherit from `Object`, a filter on this class is applied on all messages to objects. So all invocations of methods in the whole system are traced. The actual filter method displays the calls and exits of methods with an according message. The CALL traces are supplied with the arguments, the EXIT traces contain the result values. We have to avoid the tracing of the trace methods explicitly. With a more sophisticated filter implementation, the trace can be restricted to instances of certain classes, or produce trace output for only certain methods.

```
Object instproc traceFilter args {
  # don't trace the Trace object
  if {[self] == "::Trace"} {return [next]}
  ::set method [[self] info calledproc]
  puts "CALL > [self]->$method $args"
  ::set result [next]
  puts "EXIT > [self]->$method ($result)"
  return $result
}
Object filter traceFilter
```

Related Work The underlying idea behind filters (and per-object mixins) are interceptors for messages. In [1] objects that are able to abstract interactions among objects are introduced. The message passing model is enhanced by input/output interception, an idea generally introduced in CLOS [4].

Our driving motivation for the implementation of filters was language support for design patterns [25]. Several authors proposed other reflection and interception mechanisms for their implementation of design patterns. The LayOM-approach [5] is the most similar to the filter approach. It offers an explicit representation of patterns using an extended object-oriented language. The approach is centered on message exchanges as well and puts layers around the objects which handle the incoming messages. The filter approach differs from LayOM since it can represent patterns as ordinary classes and needs no new constructs, only regular methods. The FLO-language [11] introduces a "component connector" that is placed between interacting objects. Connectors are controlled through a set of interaction rules that are realized by operators. Hedin [17] presents an approach based on an attribute grammar in a special comment marking the pattern in the source code. The comments assign roles to the classes, which constrain them by rules. Constraining of patterns can be achieved in XOTcl using assertions (see Section 6), which can be checked at run-time.

4 Per-Object Mixins

Per-object mixins are a novel approach of XOTcl to extend the method chaining of a single object. Therefore, per-object mixins can handle complex data-structures dynamically on a per-object basis. The term "mixin" is a short form for "mixin class".

Definition 2 *A per-object mixin is a class which is mixed into the precedence order of an object in front of the precedence order implied by the class hierarchy.*

An arbitrary class can be registered as a per-object mixin for an object by the predefined `mixin` method. This method accepts a list of per-object mixins for the registration of multiple mixins. The following example defines the classes `Agent` and `MovementLog` (each with a same-named method) and registers `MovementLog` on the `Agent`-instance `a` as a mixin:

```
Class Agent
```

```
Agent instproc moveAgent {x y z} {
  puts "moving"
  # do the movement ...
}
Class MovementLog
MovementLog instproc moveAgent {x y z} {
  puts "Agent [self] moves to ($x,$y,$z)"
  next
}
Agent a -mixin MovementLog
```

Per-object mixins use the `next`-primitive to forward messages to the chain of other mixins and to pass it finally to the ordinary class hierarchy of the object. If a call on object `a` is invoked, like "`a moveAgent 1 2 3`", the per-object mixin is mixed into the precedence order of the object, immediately in front of the precedence order resulting from the class hierarchy (as illustrated in Figure 4). The resulting output of the example call is:

```
moving
Agent a moves to (1,2,3)
```

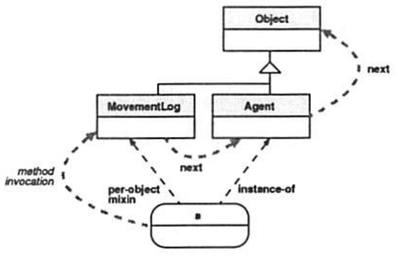


Figure 4: Per-Object Mixin Example

Mixins can be removed dynamically at arbitrary times by handing the `mixin` method an empty list. Methods of mixins have full access to all introspection options. As interceptors they additionally have access to the `info`-options `callingproc`, `callingclass` and `callingobject` (see Section 3). The registered mixins can be introspected using

objName `info mixins` *?class?*

which returns the list of all mixins of the object, when `class` is not specified. Otherwise it returns `1`, if `class` is a mixin of the object, or `0` if not.

The `mixin` relationship of an object can be used to model extrinsic properties (such as roles), whereas the `class` relationship is used to define its intrinsic properties. Per-object mixins are ordinary classes and support full specialization/generalization to make their usage compatible with ordinary classes.

Related Work Per-object mixins use the same mechanism as the method chaining in OTcl [32]. Both the precedence order and the idea of mixins in OTcl are influenced by the lisp extension CLOS [4, 18]. The filter approach (discussed in last section) is the class-level interception construct whereas per-object mixins are interceptors for single objects independent of the class relationship. A comparison between per-object mixin and filter can be found in [26]. In Agora [30] mixins are treated as named attributes of classes. In [7] different inheritance mechanisms are compared and mixins are proposed as a general inheritance construct.

Modeling of objects changing roles (as in [14]) can be implemented through the change of class relationships (see Section 2). In [24] we provide a deeper discussion of per-objects mixins and we point out that the per-object mixin are well-suited to model roles. Per-object mixins are transparent to clients. They let us decompose extrinsic (role) and intrinsic properties of objects into classes and combine them into one conceptual entity. Bosch proposes in [6] a component adaption technique, which is similar to the per-object mixin idea. It is also transparent, composable and reusable, but it is not introspective, not dynamic and a pure black-box approach.

5 Nesting of Classes and Objects

Most object-oriented analysis and design methods are based on the concepts of generalization and aggregation [29]. Generalization is achieved through class hierarchies and inheritance, while (static) aggregation is provided through embedding.

In order to support (static and dynamic) aggregation we use the namespace concept provided by Tcl since version 8.0. A *namespace* provides an encapsulation of variable and procedure names in order to prevent unwanted name collisions with other system components. Each namespace has a unique identifier which becomes part of the fully qualified variable and procedure names. Namespaces are therefore already object-based in the terminology of Wegner [31]. OTcl is object-oriented since it offers classes and class inheritance. Since objects in OTcl provide namespaces (with different semantics) as well, two incompatible namespace concepts existed in parallel. In OTcl every object has a global identifier.

XOTcl combines the namespace concept of Tcl with the object concept of OTcl. Every object and every class in XOTcl is implemented as a separate Tcl namespace. The biggest benefit of this design decision aside from performance advantages is the ability to construct aggregated objects/nested classes and to reduce name conflicts. Note, that the namespaces do not eliminate all possible naming conflicts. In XOTcl object identifiers are Tcl commands. In the case of nested objects, a name conflict between the object names and per-object procs may arise.

Through the strong integration with the Tcl namespaces we achieved additional advantages: Instance variables are traceable in XOTcl through Tcl's `trace` command, and methods may be executed via namespace qualification directly (by-passing XOTcl's dispatch), which can make method invocation as fast as Tcl's proc invocation for performance critical sections (although loosing the assertion and interception facilities of XOTcl).

Nested Classes As a simple example of nested classes, the description of a oval carpet and a desk can nest inside a `OvalOffice`-class:

```
Class OvalOffice
Class Carpet                ;# a general carpet
Class OvalOffice::Desk
# special oval carpet - no name collision
Class OvalOffice::Carpet -superclass ::Carpet
```

Nested classes have the same properties as ordinary classes. Additionally the information aboutthe nesting is available through the `info` method:

> *ClassName* `info classchildren`
> *ClassName* `info classparent`

The `classchildren` option returns a list of children (possibly empty). `classparent` results in the name of the parent class, if the class is nested. In order to ease the construction of the full path of a namespace we support the following two alternative syntax forms for the creation of nested classes:

> `MetaClassName` *ClassName::nestedClass*
> *ClassName* `MetaClassName` *nestedClass*

Dynamic Object Aggregation Every object in XOTcl has its own namespace which can contain other objects. Suppose an object of the class `Agent` should aggregate some property objects of an agent, such as head and body:

```
Class Agent
Class Agent::Head
Class Agent::Body
```

The classes `Head` and `Body` are in the `Agent` namespace and do not infer with other same-named classes.

```
Agent myAgent
Agent::Head ::myAgent::myHead
Agent::Body ::myAgent::myBody
```

Now `myHead` and `myBody` are part of `myAgent` and they are accessible through the full namespace path. These paths can turn out to be quite cumbersome to write. Fortunately, in most situations a programmer does not have to write the full path, since within XOTcl methods the object's namespace is set automatically and the `self` command obtains the fully qualified object name. For the creation of aggregated objects the following two forms can be used:

> `ClassName` *objName::aggregatedObj*
> *objName* `ClassName` *aggregatedObj*

The information about the part-of relationship of objects can be obtained the same way as for classes through the `info` method interface:

> *objName* `info children`
> *objName* `info parent`

Dynamic Aggregation It is likely that all agents have properties for head and body. This implies a static or pre-determined relationship between class nesting and object aggregation. A pre-determined aggregation of property objects can be built through a constructor, such as:

```
Agent instproc init args {
  ::Agent::Head [self]::myHead
  ::Agent::Body [self]::myBody
}
Agent myAgent
```

Every agent is now created with a head and a body. The aggregation can be changed dynamically at runtime by creation or destruction of objects. The `destroy` method turns the agent into a headless agent:

```
myAgent::myHead destroy
```

XOTcl provides introspection for aggregations as well. Suppose, that in the virtual world the agents heads may be slashed from their bodies. The graphical system can simply ask the agent with `info children`, whether it has a head or not, and can choose the graphical representation accordingly.

Every object/class can be moved (and copied) to an other object/class by the `move` (`copy`) method. These are deep operations, effecting the object and all its aggregates.

Related Work Our view of aggregation is influenced by investigations on modules [13], conceptual modeling [29], and object-oriented languages, like Troll [15]. Several common programming languages offer nested or inner classes, e.g. Java, C++, Beta, etc. These concepts provide aggregation of descriptive structures, but lack in introspection abilities and dynamics. Banavar [3] points out that class-level nesting is a form of composition.

In languages without support for object aggregation, it is approximated by embedding of objects or by association through pointers (a reference). Embedding does not permit dynamic aggregations, pointers lead to low level programming with hand coded operations for operators like deep-copy/-move. These approximations contradict the idea of an aggregation.

Several design patterns also use aggregations for composition. The whole-part pattern [9] aggregates objects of arbitrary types. Dynamic object aggregations form a language support for this pattern. The more special variant "composite" [12] aggregates hierarchies of objects of the same type and can also be language supported (see Section 7).

6 Other Functionalities

This section describes briefly some other language functionalities that are not available in OTcl.

Abstract Classes A class is defined abstract if at least one method of this class is abstract. The build-in method `abstract` defines an specifies the interface of an abstract method. Direct calls to abstract methods produce an error message. E.g. a `Graphic`-class provides an abstract interface for drawing:

```
Class Graphic
Graphic abstract instproc draw args
```

Parameters Classes may be equipped with `parameters` definitions which are automatically created for the convenient setting and querying of instance variables. Parameters may have a default value, e.g.:

```
Class Person -parameters {
  name
  {friends ""}
  {ID [self]}
}
```

Each instance of class `Person` has three properties defined. `name` has no default value, `friends` defaults to an empty list, and the `ID` defaults to the instance's self-ID. `parameters` are inherited to subclasses. The following example demonstrates setting and querying of parameters:

```
Person p1 -name Anakin ;#set name at creation
p1 name "Darth Vader"  ;#set name at runtime
puts "Name of p1: [p1 name] objID: [p1 ID]"
```

Assertions In order to improve reliability and self documentation we added assertions to XOTcl. The implemented assertions are modeled after the "design by contract" concept of Bertrand Meyer [21, 22]. In XOTcl assertions can be specified in form of formal and informal pre- and post-conditions for each method. The conditions are defined as a list of and-combined constraints. The formal conditions have the form of ordinary Tcl conditions, while the informal conditions are defined as comments (specified with a starting "#"). Pre- and post-conditions are appended as lists to the method definition.

Since XOTcl offers per-object specialization it is desirable to specify conditions within objects as well (this is different to the concept in [21]). Furthermore there may be conditions which must be valid for the whole class or object at any visible state (that means in every pre- and post-condition). These are called invariants. Logically all invariants are appended to the pre- and post-conditions with a logical "and". The syntax for class invariants is:

ClassName `instinvar` *invariantList*

and for objects invariants:

objName `invar` *invariantList*

All assertions may be introspected. Since assertions are contracts they need not to be tested if one can be sure that the contracts are fulfilled by the partners (see [21]). But for example when a component has changed or a new one is developed the assertions could be checked on demand. The checking is then fulfilled at the beginning and at the end of each method call. The `check` method has configuration options for all assertions types to turn checking on/off. The syntax is:

objName check ?all? ?instinvar? ?invar? ?pre? ?post?

Per default all options are turned off. `check all` turns all assertion options for an object on, an arbitrary list (maybe empty) can be used for the selection of certain options. Assertion options are introspected by the `info check` option. The following class is equipped with assertions:

```
Class Sensor -parameters {{value 1}}
Sensor instinvar {
  {[regexp {^[0-9]$} [[self] set value]] == 1}
}
Sensor instproc incrValue {} {
  incr [self]::value
} \
{{# pre-condition:} {[[self] value] > 0}}  \
{{# post-condition:} {[[self] value] > 1}}
```

The parameter instance method defines an instance variable value with value 1. The invariant expresses the condition (using the TCL command regexp), that the value must be a single decimal digit. The method definition expresses the formal contract between the class and its clients that the method incrValue only gets input-states in which the value of the variable value is positive. If this contract is fulfilled by the client, the class commits itself to supply a post-condition where the variable's value is larger than 1. The formal conditions are ordinary TCL conditions. If checking is turned on for sensor s:

```
s check all
```

the pre-conditions and invariants are tested at the beginning and the post-condition and invariants are tested at the end of the method execution automatically. A broken assertion, like calling incrValue 9 times (would break the invariant of being a single digit) results in an error message.

We have already pointed out that the presented concepts are relying on Meyer's Design by Contract [21, 22]. The differences are, that due to the ability to define per-object specializations object assertions are introduced, and that due to the dynamics of the language the assertions are also dynamically changeable and introspectable.

AsserTcl [10] is another approach that introduces assertions into TCL through four new TCL commands. The advantages of our approach are the integration with object-orientation , the placement of assertions at a familiar place (at the end of the method definition) and the support for invariants in classes and objects.

Meta-data To enhance the self-documentation and the consistency between documentation and program it is useful to make the documentation a part of the program, i.e. to store meta-data like the author, a description, the version, etc. Meta-data registered for classes is inherited and propagated to all instances and are a dynamic and introspective. Syntactically, meta-data can be specified through the metadata method with its options add and remove. Other arguments for the metadata method are interpreted as meta-data values. E.g. on Object:

```
Object metadata add {description version}
Object metadata description "This class \
  realizes all common object behavior"
Object metadata version "1.0"
```

Since the meta-data registered on classes is inherited, all objects can store information on description and version. E.g. the agent a1 stores such values:

```
Agent a1
a1 metadata description "My testing agent"
a1 metadata version "0.1"
```

Classes and objects can store additional meta-data for their own purposes at any time. E.g. agents can store information about the host on which they have been created. Introspection of meta-data is implemented through info metadata method, which lists all defined meta-data attributes with values associated (only those). Meta-data values can be accessed through the metatdata instance method, similar to the usage of set. If no value parameter is given, the current value is returned. The following command produces the result "1.0":

```
Object metadata version
```

Beneath the special features, like inheritance, meta-data could be expressed through instance variables solely. But especially in distributed environments, it is important to have such a facility, because the common place and the introspection mechanisms allow different people and, in XOTCL's case, even other programs, to gain the meta-data without searching for them. Since this is not only a naming convention, but a language construct, the interpreter results in an error if the meta-data is used incorrectly.

Automatic Name Creation The XOTCL autoname instance method provides an simple way to take the task of automatically creating names out of the responsibility of the programmer. The example below show how to create on each invocation of method new an agent with a fresh name (prefixed with agent):

```
Agent proc new args {
  eval [self] [[self] autoname agent] $args
}
```

7 Application Example: An XML-Parser/-Interpreter

Now we illustrate the usage of the new language constructs in a larger example: the design pattern based architecture/implementation of an XML-Parser/-Interpreter. XML [8] is a meta-language that defines on basis of a document type definition (DTD) the structure of an application document.

An application program that wants to extract information from an XML document has to parse the document and has to interpret the resulting structure. The (partial) design of our implementation is presented in Figure 5, where a wrapper facade pattern integrates and encapsulates an off-the-shelf XML parser, the interpreter-/composite-patterns abstracts from the syntax tree representation, a builder separates the parsing from the creation of the resulting structure, a visitor decouples the interpretation from the syntax tree and a per-object observer is used to trace visitations.

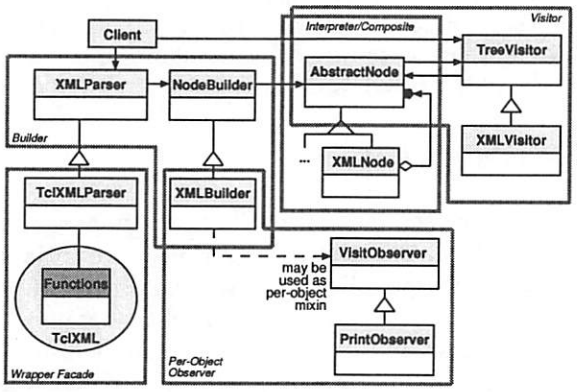


Figure 5: Partial Design of the XML Parser/Interpreter

The interpreter pattern [12] defines an object-oriented representation for a grammar along with an interpretation mechanism (essentially an interpret method for each node type). All clients abstract from expressions through the use of an abstract interface for interpretation purposes. At run-time expression objects form an abstract syntax tree. Frequently the interpreter's tree representation is implemented as a composite pattern, that arranges objects in a tree with leaf and composite nodes of the same component type. Registered composite operations are recursively forwarded to all children of the composite.

The composite pattern can be implemented through a meta-class that registers a filter in the constructor of the meta-class for forwarding of the operations to the nested nodes. We implement the abstract pattern in a meta-class `Composite` which can be loaded from a library and reused and instantiated in several application classes. `addOperations` registers and `removeOperations` unregisters the operations which should be forwarded to the nested nodes.

```
Class Composite -superclass Class
Composite instproc addOperations args {...}
Composite instproc removeOperations args {...}
Composite instproc init args {...}
```

The registered operations are stored in the associative array `ops` (a class variable in the class on which the filter was registered) that is accessed by a generic filter which performs the actual forwarding:

```
Composite instproc compositeFilter args {
  set m [[self] info calledproc]
  set c [[self] info regclass]
  set r [next]
  if {[info exists ${c}::ops($m)]} {
    foreach child [lsort \
      [[self] info children]] {
      eval [self]::$child $m $args
    }
  }
  return $r
}
```

The filter determines through `info calledproc` the method which is called, obtains the registration class of the filter through `info regclass`, and checks, whether the called method `m` was registered in the array `ops` for forwarding. If `m` is registered, the message is passed to all children objects (determined by `info children`). Finally, `next` forwards the message to the actual node. Since the children may be composites as well, this mechanism iterates recursively on the entire tree structure. The filter is registered for derived application classes in the constructor of `Composite`.

Now the meta-class `Composite` can be used to implement the interpreter pattern for XML-nodes. We define a class `AbstractNode` with the meta-class `Composite` and an abstract operations for accepting interpretation through a visitor, which should work recursively, therefore it is added as a composite operations. Application node classes such as `XMLNode` can be derived by specializing `AbstractNode`:

```
Composite AbstractNode
AbstractNode abstract instproc accept v
AbstractNode addOperations accept
Class XMLNode -superclass AbstractNode
```

For parsing of XML documents several parsers (such as TclXML [2]) can be used. In order to use a legacy parser in an exchangeable manner it has to be encapsulated transparently. The wrapper facade pattern [28] provides a general means to shield clients from direct dependencies to functions.

Actually we use `XMLParser` as a wrapper facade object that embodies the interface to the XML parser as object-oriented methods. `configure` sets the parser configuration, `cget` queries the configuration, `parse` invokes the parsing of XML text, and `reset` cleans up the parser context before a new text can be parsed.

```
Class XMLParser -parameters {nodeBuilder}
XMLParser abstract instproc init args
XMLParser abstract instproc cget option
XMLParser abstract instproc configure args
XMLParser abstract instproc parse data
XMLParser abstract instproc reset {}
```

A specific TclXML parser is derived by sub-classing:

```
Class TclXMLParser -superclass XMLParser
```

In order to separate the construction process of the complex node structure from its representation and to make representations exchangeable, we use the builder pattern [12]. The parser is the director of the construction process, invoked by `parse` operation. `parse` uses the `NodeBuilder` interface containing three methods to build the data representation (`startElt` actually creates nodes). A concrete implementation `XMLBuilder` builds the abstract syntax tree.

```
Class NodeBuilder
NodeBuilder abstract instproc charData text
NodeBuilder abstract instproc startElt \
  {tag attrList}
NodeBuilder abstract instproc endElt tag
Class XMLBuilder -superclass NodeBuilder
```

The visitor pattern [12] is used to define operations on the nodes independently from the intrinsic properties of the nodes. According to the pattern the nodes have to concretize an `accept` method , which calls the `visit` method of the visitor for all nodes (since it is registered as a composite method):

```
XMLNode instproc accept v {$v visit [self]}
```

Below is an abstract `TreeVisitor` and a specialization `PrintVisitor` that just prints out every node:

```
Class TreeVisitor
TreeVisitor abstract instproc visit objName
Class PrintVisitor -superclass TreeVisitor
PrintVisitor instproc visit objName {
  puts "Visitation of node $objName"
}
```

Such visitors can be used e.g. for locating XML-elements with certain properties, for extracting or mapping of the XML-structure, etc. For observing certain events in the system, a per-object observer [26] may be used, which can for example observe the parsing of the document. The per-object observer is based on the observer in [12] and implemented through per-object mixins. The `PrintObserver` watches the processing of `startElt` tags:

```
Class PrintObserver
PrintObserver instproc startElt {t a} {
  puts stderr "... watching: name=$n"; next
}
```

The `PrintObserver` can be added to the `XMLBuilder`, which should be observed. Finally, the parser is connected with the node-builder instance:

```
XMLBuilder ::t -mixin PrintObserver
TclXMLParser x -nodeBuilder ::t
x parse "...<PERSON>...</PERSON>..."
```

The implementation of the presented XML processing system has the size of 73 lines (including two more useful visitors) plus 22 lines for the wrapper facade, and 25 lines for the implementation of `Composite` (including the definitions of abstract interfaces).

8 Conclusion

This paper introduces XOTcl, which is an experiment to combine the benefits of a scripting language with the benefits of a high level object-oriented language. We tried to preserve the underlying principles of the scripting language Tcl (like dynamic typing, flexible glueing of preexisting components, read/write introspection) while extending the language with high level object-oriented concepts (like filters, per-object mixins and dynamic aggregations). Our goal was to improve productivity and software reuse by providing the programmer with powerful means to manage complexity and to improve composability.

We used the new language concepts with promising success in various applications, including a web browser [19], an HTTP server, a web-based object and a mobile code system [34], a persistent store, an XML-/RDF-Parser, etc. The new language constructs helped to improve the modularization, the robustness and the code size of these systems.

XOTcl is available from:
`http://nestroy.wi-inf.uni-essen.de/xotcl.`

References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. LCNS 791, Springer-Verlag, 1993.

[2] S. Ball. TclXML. http://www.zveno.com/zm.cgi/in-tclxml/, 1999.

[3] G. Banavar. Nesting as a form of composition. In *Proc. of CIOO Workshop at ECOOP*, July 1996.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *Sigplan Notices*, 23(9), 1988.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 11(2):18–32, 1998.

[6] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41, 1999.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA/ECOOP'90*, volume 25 of *SIGPLAN Notices*, pages 303-311, October 1990.

[8] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language. http://www.w3.org/TR/REC-xml, 1999.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture – A System of Patterns.* J. Wiley and Sons Ltd., 1996.

[10] J. Cook. Assertions for the TCL language. In *Proc. of the Fifth Annual Tcl/Tk Workshop 1997*, Boston, 1997.

[11] S. Ducasse. Message passing abstractions as elementary bricks for design pattern implementations. In *Proc. of LSDF'97*, 1997.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[14] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), 1996.

[15] T. Hartmann, R. Junghans, and G. Saake. Aggregation in a behavior oriented object model. In O. Madsen, editor, *Object-Based Distributed Processing*, pages 57–77. LCNS 615, Springer-Verlag, 1992.

[16] L. Hatton. Does OO sync with how we think? *IEEE Software*, May/June 1998.

[17] G. Hedin. Language support for design patterns using attribute extension. In *Proc. of LSDF'97*, 1997.

[18] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[19] E. Köppen, G. Neumann, and S. Nusser. Cineast – an extensible web browser. In *Proc. of the WebNet 1997 World Conference on WWW, Internet and Intranet*, Toronto, Canada, November 1997.

[20] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory & practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996.

[21] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[22] B. Meyer. Building bug-free o-o software: An introduction to design by contract. http://eiffel.com/doc/manuals/technology/contract/index.html, 1998.

[23] G. Neumann and S. Nusser. Wafe – an X toolkit based frontend for application programs in various programming languages. In *Proc. of USENIX Winter 1993 Technical Conference*, San Diego, January 1993.

[24] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. Proc. of Asia-Pacific Software Engineering Conference (APSEC), December 1999.

[25] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proc. of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, May 1999.

[26] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proc. of NOSA'99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.

[27] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31, March 1998.

[28] D. C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report, SIGS*, 11(2), February 1999.

[29] J. Smith and D. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), 1977.

[30] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in Agora. In *Proc. of ECOOP '93*, LNCS 707. Springer-Verlag, 1993.

[31] P. Wegner. Learning the language. *Byte*, 14(3):245 – 253, March 1989.

[32] D. Wetherall and C. J. Lindblad. Extending TCL for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.

[33] U. Zdun. Entwicklung und Implementierung von Ansätzen, wie Entwurfsmustern, Namensräumen und Zusicherungen, zur Entwicklung von komplexen Systemen in einer objektorientierten Skriptsprache. Diplomarbeit (diploma thesis), Universität Gesamthochschule Essen, 1998.

[34] U. Zdun. Entwurf und Entwicklung eines mobilen Objekt-Systems für Anwendungen im Internet. Diplomarbeit (diploma thesis), Universität Gesamthochschule Essen, 1999.

A Multi-threaded Server for Shared Hash Table Access

Andrej Vckovski and Jason Brazile
Netcetera AG
{andrej.vckovski,jason.brazile}@netcetera.ch

Abstract

This paper presents a multi-threaded socket server allowing access to shared hash tables. It is implemented using Tcl 8.1 multi-threading capabilities and runs multiple Tcl interpreters to service client requests. The application is designed as a pre-threaded server which allows a single working thread to handle many requests. The central shared data object is a hash table with structured values which allows access by all threads. Synchronization is based on a reader/writer lock implementation using the synchronization primitives available in Tcl, i.e., mutexes and condition variables. The application achieves insert rates that are significantly higher than what current commercial database management systems achieve. The usage of third-level language programming in C and application-specific scripting in Tcl allows a design based on a light-weight, robust kernel on the one hand and easily modifiable application-domain code on the other. The experiences with thread-safety and other threading features in Tcl 8.1 have been largely positive in this real-world application.

1 Introduction

There are two motivating factors for presenting this system. First, it provides an example of the usage of some of the multi-threading capabilities introduced with Tcl 8.1. Second, it provides a general solution to the frequent desire to have shared, direct access, tabular data in the context of transaction-oriented applications such as programs with an HTML graphical user interface.

The original motivation for this application, however, differs slightly. The initial purpose was to support a data feed handler to provide a cache for pseudo-real-time financial market information. We describe the initial requirements in a first section, which is followed by a discussion of the system architecture. Then, we give an overview of two central implementation aspects: a pre-threaded socket server and synchronized access to Tcl hash tables. In the final section, we discuss our experiences with the Tcl threading API and some performance results.

2 Application background

Several large data vendors such as Reuters, Bloomberg, Bridge and others provide numerous kinds of financial market information, such as stock quotes, using various formats and mechanisms. Also, many stock exchanges worldwide provide direct access to their data using various formats and mechanisms. A common characteristic of all these data providing mechanisms is that the data is delivered as a sequence of incremental updates to some base information. Such incremental updates usually contain a unique ID identifying the item being updated and then a set of key/value pairs providing the new attributes of that item. Such attributes might be for example the last paid price and its associated time stamp for a given financial instrument.

There are basically two typical access patterns to that sort of information. A user either requests all or selected attributes of a financial instrument (request/response) or the user subscribes to a financial instrument with the intention to receive all further updates to that item (subscribe/notify). Since the update rates on data feeds can be quite high (several hundred update messages per second) it is common to store all information in main memory, especially as memory cost currently allows main memory sizes of several gigabytes. Thus, such main memory caches need to support insertion of incremental updates on the one hand while being able to answer user queries for current data on the other.

The application presented here needs to handle various data feeds as described above. Therefore, it was necessary to provide an environment that allows quick change cycles. If 6 different data formats have to be supported and every data format definition changes once a year, the overall change rate might be one change every 2 months. Also, the feed handler, being a server or daemon process, needs to be very robust because it will have very long run times and be accessed by many clients. For the implementation, there were basically three implementation alternatives:

- Use a commercial (disk-based) database management system (DBMS) that can be tuned to cache tables entirely in main memory (for performance reasons)
- Use a commercial main-memory database (MMDB)
- Develop a specific cache manager

The first two options had been rejected because of price/performance issues (e.g., an RDBMS that has to provide these update rates is very expensive to build and maintain) as well as robustness and complexity (MMDB) issues.

Also, the typical data structure can not be efficiently described as relations. There are many possible attributes for every item (instrument), but in most cases, only a few of them actually have values, i.e., most 'columns' are 'null'. In addition, frequent changes to the set of attributes are made, usually consisting of the addition of new attributes. Therefore, a relational approach would be either computationally expensive if the attributes would need to be normalized in a second table, or hard to maintain if the database structure had to be permanently modified. Based on these arguments, we decided to build a custom feed handler and cache management system using as much existing technology as possible and focusing on simplicity, robustness and maintainability.

3 System architecture

Based on the requirements discussed above, we chose an implementation strategy consisting of:

- A multi-threaded socket server
- Using Tcl's hash tables for data storage
- Using one Tcl interpreter per servicing thread to provide easy customization of data manipulation

The idea was that the server would run a thread for every connected client and maintain a set of shared hash tables. Every thread would have its own Tcl interpreter with which the clients would communicate, and the interpreter would have special commands that allow clients to access the shared hash tables, i.e., insert, retrieve and delete keys and values. The server should only provide very general functionality and let users or client applications implement most higher level functionality using Tcl code rather than more error prone C or C++ programming.

This application needs to run on Unix platforms and so we chose C, Tcl 8.1 and POSIX threads as the implementation basis. C++ was considered somewhat fragile because in multi-threaded applications it is always very important that one knows exactly what is going on to prevent race conditions and deadlocks. C++ compilers and runtime libraries tend to include hidden overhead into the application code that cannot be easily tracked. Also, we wanted to reduce overall complexity as much as possible.

The choice of a multi-threaded server allows for an easy usage of shared data structures, e.g. as in the hash tables mentioned above. However, unlike a forking server that spawns off separate processes to service requests, a multi-threaded server is less robust in the sense that there is no address space isolation preventing problems in servicing a request that can impair overall stability. Still, we believe that by making a server as generic as possible, we can reduce the complexity to a level where it is possible to design, implement and test a fairly stable multi-threaded server.

The communication with clients is based on TCP using Berkeley sockets with a simple message format. A request message consists of Tcl code that is evaluated in the thread's Tcl interpreter. The message result is either the result of the evaluated Tcl code or the corresponding error message. That is, the model is similar to Tk's `send` command.

The application's main thread also has a Tcl interpreter. This interpreter is "connected to" standard input and output (i.e., takes commands from standard input) and has a few additional commands that support creation of the server socket, maintaining the number of current threads and doing signal handling. Signal handling is an important requirement for long-running processes and is discussed in a separate section below.

In addition to threading and hash table extensions, every Tcl interpreter (i.e., the per-client thread interpreters and main thread interpreter) have additional commands for logging. Similar to signal handling, useful logging with several levels of verbosity is also very important in long running processes because such programs usually have no user interface and therefore rely on log files for their output. Signals and log files can be seen as a kind of primitive user interface to daemon processes.

The architecture described above is shown in figure 1.

The next section covers a few aspects of the implementation.

4 Shared hash tables and locking

Most objects being used by multiple clients at the same time have a need for some kind of synchronization. Sometimes that need is even the key motivation for the object to be used by multiple clients. In our case, the synchronized objects are hash tables that can be accessed by all threads. The synchronization has to guarantee consistency of the hash tables with parallel inserts, updates, and queries on the data. The selection of a synchronization mechanism usually depends strongly on the expected access patterns which can significantly influence system performance if, for example, an implementation variant is chosen that leads to high lock contention. The main synchronization questions we faced here were similar to typical locking issues in database management systems:

- Locking granularity
- Lock sharing

Locking granularity describes the trade-off between fine-grained locking, which minimizes lock contention but needs a large number of locks to be acquired when doing operations involving many data items, and course-grained locking, which uses fewer locks at the expense of performance. For example, most commercial database management systems offer different levels of locking granularity. On full table scans single table locks are used, while cursor operations rely on a logical row-level locking or a physical page-level locking. However, such adaptive locking granularities also introduce a lot of additional complexity which - especially in the case of locking - increases the risk of errors and dead lock situations.

Lock sharing describes the situations where there is an access pattern which allows readers and writers to be distinguished. Readers can be synchronized using a shared lock (many readers can simultaneously access a resource) while writers need exclusive access to the resource. Again, there is usually a trade-off between using exclusive locks and shared locks. Shared locks are typically more expensive to acquire but reduce lock contention.

In our application, a few estimates showed us that as a first approximation, it would be sensible to use shared (reader/writer) locking on the entire hash table. That is, a reader always locks the entire table for shared access, and a writer locks the entire table for exclusive access. This approach turned out to be very efficient in our case where there is typically only one or a few writers and many readers.

The (preliminary) Tcl 8.1 threading API offers (similar to POSIX threads) two synchronization primitives: Mutexes (exclusive locks or semaphores) and condition variables. It was therefore necessary to provide our own implementation of reader/writer locks based on mutexes. Considering the frequent need for reader/writer locks and their simple implementation, it would be worthwhile to include an implementation in the Tcl API.

Our shared hash table implementation was therefore protected by a single reader/writer lock that is acquired depending on the type of operation. The hash table is based on Tcl's hash table with the extension that there is additional structure imposed on the values entered in the hash table: all values ('rows') are entered as attribute/value pairs allowing selective 'columns' or attributes to be queried and updated.

The access to shared hash tables is implemented using two command procedures. The first command called `sharedhash` is used to create, delete and use hash tables in an interpreter. After having either created a new hash table or acquired a handle to an existing hash table, a second command is created in that interpreter with the same name as the hash table. The hash table is then accessed using this command much in the same way as for example Tk widgets or [incr Tcl] instances are associated with a command, as the following example illustrates:

```
# create a shared hash table
# with the name 'foo'
sharedhash create foo 20
# use the hash table named
# 'foo' in this interpreter
```

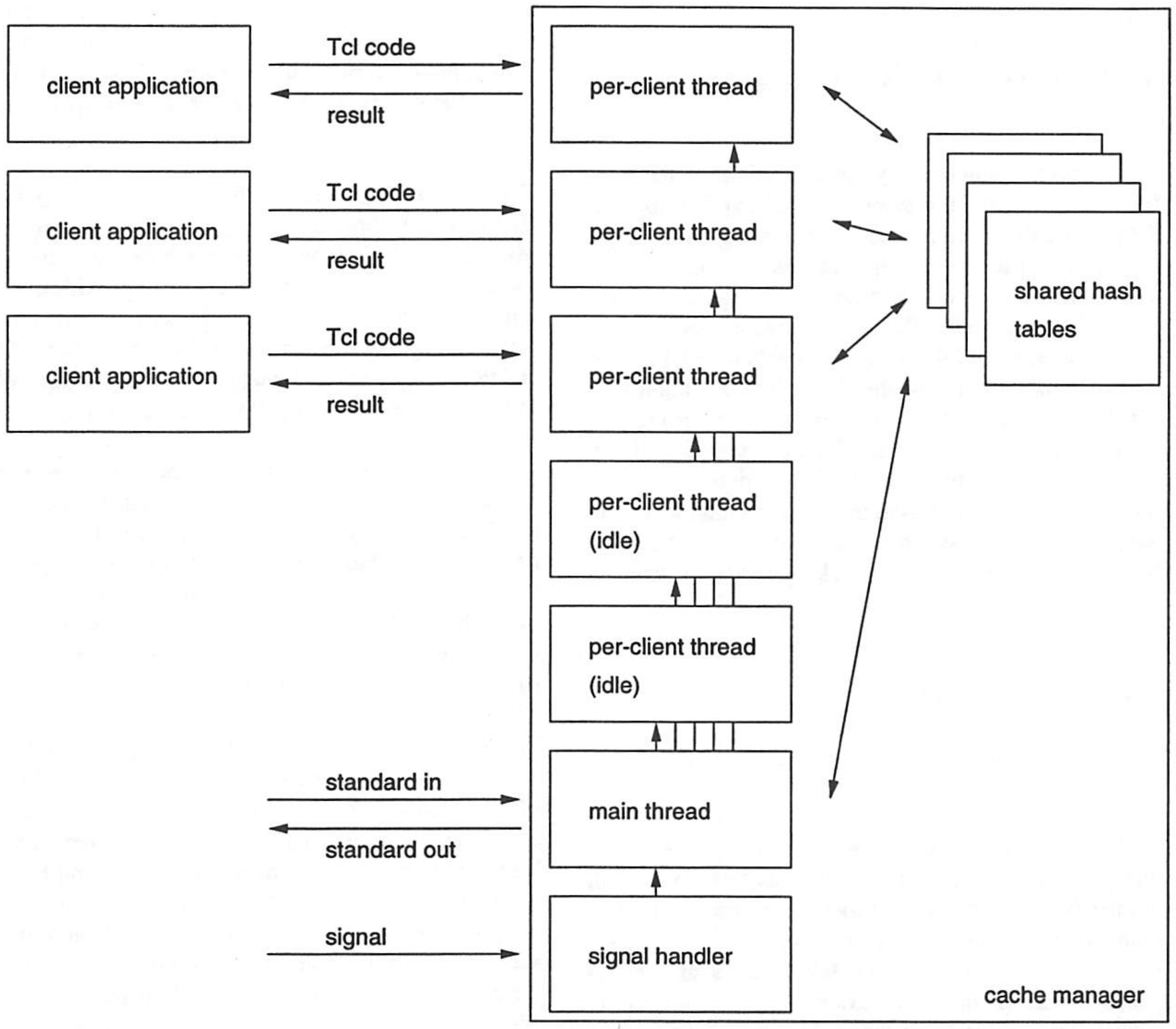


Figure 1: Architecture overview

```
shared hash use foo
# access the shared
# hash table
foo set 12345 {
  attrA valueA
  attrB valueB
}
# set an array with the
# attributes of the hash value
array set res [foo get 12345]
```

The following subcommands for the `sharedhash` command are supported:

sharedhash create ***name ?maxattrs?***

Creates a new shared hash table with the given name which can hold at most *maxattrs* attributes per row.

sharedhash use ***name***

Makes the shared hash table with the given name accessible in the current interpreter as a command with the same name.

sharedhash names

Returns a list of all defined shared hash tables

sharedhash forget ***name***

Removes a reference to the shared hash table from this interpreter, i.e., deletes the associated command. If that was the last reference, deletes the hash table.

The hash table is then accessed using the following subcommands:

name **set** ***key attribute-value-list***

Sets the *attribute-value-list* to the current content of the hash tables value associated with *key*, deleting all previously defined attributes.

name **get** ***key ?attribute-list?***

Gets either the named or all attributes for the given *key*.

name **update** ***key attribute-value-list***

Updates the current values associated with *key* with the *attribute-value-list*.

name **names** ***?pattern?***

Returns a list of keys matching the pattern or all keys

name **attributes**

Returns a list of all the currently defined attributes

name **delete** ***key***

Removes the entry from the hash table

name **foreach** ***pattern varnames attributes code***

Loops over all entries with keys matching the pattern and assigns the variable names in *varnames* the values of the attributes *attributes*.

name **updateforeach** ***pattern variablename code***

Loops over all matching entries with a write lock and executes code for every entry. The code can access the current element using a special key `#current`.

name **stats**

Returns hash table statistics as returned by `Tcl_HashStats`.

5 Pre-threaded socket server

As mentioned above, performance was a critical issue. Therefore, we decided to base the server on a pre-threaded design [1]. This means that the servicing threads are not created for every request but pooled in advance. This approach has a few advantages:

- Threads can be easily re-used. Initializations such as the creation of a Tcl interpreter needs only to be done once. A thread can service many requests, i.e., not every new request needs the creation of a new thread.
- Overall server design is more symmetrical because there is not a special thread that accepts connections and spawns off new threads.

However, there are also some drawbacks with a pre-threaded solution. As the load changes over time, it might be necessary to asynchronously create and destroy threads. While the creation is simple, the destruction of running threads is not trivial, especially if the destruction should not be deferred.

A servicing thread's main function is to perform the following steps:

1. increment the thread counter

2. create an interpreter
3. register all additional commands
4. evaluate thread constructor Tcl code
5. while (termination not needed)
    (a) acquire exclusive lock to accept a connection
    (b) wait for a incoming request (`accept`)
    (c) release lock
    (d) increment working thread counter
    (e) while (not end-of-file of socket)
        i. receive a message
        ii. evaluate the Tcl code
        iii. send the message with the result back
    (f) decrement working thread counter
    (g) check if there are threads scheduled to terminate
6. evaluate thread destructor Tcl code
7. delete interpreter
8. decrement thread counter
9. terminate thread

6 Main thread

In addition to the request handler threads, the application also runs a main thread that is different from the request handlers. The main thread executes the `Tcl_Main` procedure and handles standard input. The interpreter in the main thread contains additional commands to create a (listening) server socket and to control the number of working and free (not bound to a request) threads.

Long running applications in an UNIX environment usually need to handle signals for various reasons. For example, it might be necessary to perform cleanup after the process has been notified to terminate (SIGTERM signal), the application might want to re-read configuration information, dump internal information to log files or change log levels. In multi-threaded applications, signal handling needs special consideration as the so-called *async-safety* (meaning that a system call is safe to be interrupted by an asynchronous signal delivery) and *thread-safety* (meaning multiple threads may simultaneously call the function) are orthogonal in the sense that thread-safe calls are not necessarily async safe [2]. For that reason, the best way to handle signals is to block all signals in all threads and create a special signal handler thread that only waits for signals and uses thread synchronization methods (e.g., condition variables) to notify other threads if necessary.

Therefore, the main thread in this application creates a dedicated signal handler thread and also defines a new Tcl command that allows the main thread's Tcl interpreter to be notified by new signals. To keep the entire application simple, we decided not to use Tcl's event loop and notification mechanism for this purpose.

In addition to the shared hash table functionality, the following commands are available in the main thread's Tcl interpreter:

server *port ?nthreads ?constructor-code? ?destructor-code?*

This command creates a server socket and optionally *nthreads* servicing threads waiting for connections. It also allows constructor and destructor code blocks to be specified. These code blocks are executed in the servicing thread's interpreter after creation of the thread or before termination of the thread, respectively. This can be used, for example, to define variables and procedures, or load libraries or source code modules. This code does not use Tcl's `socket` function, in order to have more control over socket options such as address reuse.

servercontrol *?minidlethreads? ?maxidlethreads?*

If called without arguments, the current number of available and running threads is returned. If called with *minidlethreads* and *maxidlethreads*, it ensures that there are at least *minthreads* and at most *maxidlethreads* idle, i.e., non-servicing threads available.

waitforsignal *?sec? ?usec?*

Waits the given time (or forever) for any signal to be delivered to this application and returns the signal number.

Using this call, a typical code example running in the main thread might look like this:

```
#!/usr/local/bin/ncm

# create a shared hash
sharedhash create foo
```

```
# create a server socket
# and 10 servicing threads
server 6000 10 {
  sharedhash use foo
} {
  sharedhash forget foo
}

set terminate 0
while {!$terminate} {
  # wait 2 sec for a signal
  set s [waitforsignal 2]
  if {$s==15} {
    set terminate 1
  }
  # adapt number of idle threads
  servercontrol 5 20
}
```

7 Tcl and multi-threading

Multi-threading support has been on the wish list of the Tcl community for a long time. This was motivated mainly by application areas where Tcl was used as an embedded scripting language for some large software system, and where that software system was a multi-threaded application. The problem with Tcl was not so much that it did not support threading but that the Tcl library was not thread-safe itself (i.e., using non-reentrant functions, unsafe system calls etc.). Two extensions have emerged in the community to add thread-safety and eventually also thread support to Tcl. Steve Jankowski's MTTcl [3] used Solaris threads. Another extension was PtTcl or Pthreads-Tcl by Richard Hipp and Mike Cruse using POSIX threads [4]. Finally, thread-safety made its way into the Tcl core with Tcl 8.1. This is insofar important as most of the threading issues do indeed affect core components of Tcl and therefore, with Tcl 8.1, patching of the Tcl core isn't necessary anymore.

The multi-threading related issues in Tcl 8.1 are:

- Tcl core is thread safe, i.e., the tcl library can be used in multi-threaded applications
- Some internal data structures are stored as *thread specific data.* This allows, for example, every thread to have its own event queue.
- Some new API calls were introduced to create mutexes and condition variables. However, it seems at the time of this writing that this API isn't yet final.

The most notable missing features are official API calls for thread creation, scheduling and cancellation. Even though there are thread creation abstractions (e.g., to create a notifier thread), there are no externally usable (that is, declared in the external header `tcl.h`) functions for thread creation, cancelling and scheduling. For the time being, this is a wise choice because providing a platform independent abstraction would probably disarm many of the needed features of the platform-specific thread implementation. For example, POSIX threads allow a rich set of attributes for every thread that are not directly matched by Windows NT's different thread model. The decision to stick with a least common denominator is not a problem if the there are well-defined interfaces and transparent data structures to the underlying OS specific threading system. For example, it was easy to provide reader/writer-locks (based on Sun's SPLIT package [5]) using mutexes and condition variables once it was clear that the rest of Tcl did not use thread cancellation anywhere.

Our experiences with Tcl 8.1 and its multi-threading support showed that it is a safe choice to use Tcl 8.1 in multi-threaded applications. Even if none of Tcl's threading API is used (i.e., not even mutexes or condition variables), the thread-safety and usage of thread specific data for certain internal data structures is very useful if not mandatory within multi-threaded applications. However, it will be necessary in the future to provide detailed and precise documentation of those features. When writing multi-threaded applications, engineers are very much interested in knowing relevant "side effects" such as the creation of background threads and the like. Debugging multi-threaded applications is not trivial and therefore, one expects precise information on what should/will happen in API calls.

8 Conclusion and future work

The work presented here shows a successful example of using Tcl's new multi-threading features in a real-world application. Moreover, it proves again that a design approach using a small, reliable kernel written in a third-generation language and delegating most of the overall application complexity to a scripting language is a promising engineering approach. And, especially when developing multi-threaded applications, fighting complexity is the main objective of a system design in order to avoid dead locks, race conditions and the like. The entire cache manager and shared hash table implementation as described in here is less than 3000 lines of

C code (not including the Tcl library, of course).

The lightweight implementation is also a positive influence on performance. On a Sun 270 MHz UltraSPARC II processor, up to 1000 inserts per second can be achieved with simultaneous queries. The application went into production in late Summer 1999 in continuous (7x24 hour) operation and has not posed any major problem since. The application handles more than 1 million updates per day with peak rates of several hundred updates per second. The built-in Tcl hash function in Tcl's hash table implementation proved to be very effective, maintaining a short search distance and overall balance even in tables with more than 100,000 entries.

Our future plans with the cache manager are to provide better checkpointing (currently, the hash tables are periodically written to disk by a special client process though no consistency is enforced) and more synchronized elements among the servicing threads, e.g., named message queues. Eventually, we plan to release the code into the public domain.

9 References

1. Stevens, W. Richard. *Unix Network Programming (Vol 1)*. Prentice Hall, 2nd ed., 1997.
2. Lewis, Bill, and Berg, Daniel J. *Threads Primer*. Prentice Hall, 1995.
3. *MTtcl - Multi-threading for Tcl* `<http://www.activesw.com/people/steve/mttcl.html>`
4. *An Introduction To Pthreads-Tcl* `<http://www.hwaci.com/sw/pttcl/pttcl.html>`
5. *Solaris to POSIX Interface Layer for Threads (thread.c, thread.h and synch.h)* `<http://www.sun.com/workshop/threads/apps.html>`

Feather: Teaching Tcl objects to fly

Paul Duffin
IBM Hursley Laboratories, UK
pduffin@hursley.ibm.com, http://purl.oclc.org/net/pduffin/home

December 16, 1999

Abstract

Feather is a set of extensions which enhances the capabilities and flexibility of Tcl objects, builds a framework around them to enable new data types to be easily added and also provides a rich set of new data types. This paper starts off by giving quite a detailed description of how Tcl objects work and then moves on to describe the features of **Feather**, how they work and what they can be used for.

1 Introduction

One of the major complaints raised against Tcl by advocates of other scripting languages such as Python, Lisp and Perl is that it has very few data structures.

Feather is the remedy.

The introduction of Tcl objects in Tcl 8.0 not only greatly improved the scalability of Tcl in terms of speed and memory usage, it also provided the foundation for **Feather** to build on.

Feather is a set of extensions which not only provides lots of new data types, it also provides frameworks into which new data types can be easily inserted. It was decided early on that **Feather** must not require any changes to the Tcl core in order to make it as easy as possible for people to use. At times it seemed that the decision might have to be changed, but in the end a core change was not needed.

As there is not enough space here to describe all the features of **Feather** completely, this paper aims to provide enough information to prove that it is technically feasible and to show you some of the exciting things which **Feather** makes possible.

The paper starts off by giving quite a detailed description of how Tcl objects work and their limitations and restrictions. This should provide a sound basis for the next few sections which describe the various enhancements that **Feather** makes. The final sections describe what **Feather** uses these enhancements for, gives a brief summary of what could not be included and a look towards the future.

2 Tcl objects

This section aims to give a description of how Tcl objects work, it describes why they were introduced, gives an overview of them and then goes into more details, especially of the subtleties which cause the most problems for users.

2.1 History

Versions of Tcl prior to 8.0 used C NUL terminated character strings as their primary data type. This approach has the following scalability problems and data representation limitations.

- Everything from lists to procedure bodies to loop bodies had to be parsed every time they were used.
- Strings had to be copied whenever a long lasting reference to it was required.
- Binary data was not easily supportable because in general no length was associated with the strings.

Hence the Tcl objects were created.

2.2 Basic characteristics

The starting point for Tcl objects is that as far as the Tcl programmer is concerned they must behave exactly like strings do. In other words the Tcl programmer must be able to treat *everything as a string*.

In order to provide the necessary functionality each Tcl object has the following information associated with it.

- Reference count.
- Type.
- Dual representations
  - A string representation, including length.
  - A type specific, or internal representation.

2.2.1 Reference counted

Much of the string copying done in older versions of Tcl was done because a private reference to the data was required and there was no way of protecting the data from being changed, or of detecting when it needed to be freed.

Tcl objects solve both of these problems by using a simple reference count to keep track of how many references there are. The object can only be freed when the reference count reaches 0, and it can only be changed if the reference count is less than or equal to 1, i.e. only the code which is changing the object has a reference to it.

```
set a "hello"
set b $a
```

In the above code `a` is a reference to the literal object `"hello"` which has a reference count of 1. The assignment of `$a` to `b` simply makes `b` a reference to the object and increments its reference count to 2. The equivalent code in a string based version of Tcl would have to copy the string value of `a`.

2.2.2 String representation

This is the most important information stored in a Tcl object as it allows it to behave like a string. The length of the string is also stored in the object which means that it can handle binary data with embedded NULs.

2.2.3 Typed

A Tcl object's type basically determines the format of the internal representationn; the actual type which an object has is determined by the context it is used in. For example, an object which is used as a number will have a number type and an object which is used as a list will have a list type. If an object does not have the correct type for the current context it is automatically converted to the correct type if possible, otherwise an error is reported.

Literal strings which are found in source code have no internal type at all so they are untyped.

```
set a "1 2 3 4"
set b [lindex $a]
```

In the above code `a` starts as a reference to the literal object `"1 2 3 4"`, `[lindex]` then changes the type of the object to **list** generating an internal list representation in the process. The string representation is not changed.

Each type has to define the following operations:

- Free the internal representation.
- Duplicate the internal representation.
- Convert to type.
- Update the string representation.

2.2.4 Type specific representation

As its name suggests the nature of this information depends on the type of the object.

2.3 Behaviour and rules

Although Tcl objects are quite simple structures the rules governing their behaviour can be a little confusing. When Tcl objects were first introduced it took quite a long time to remove all of the problems caused by their unforeseen behaviour.

2.3.1 Dual representation

As mentioned above Tcl objects have two representations, a string one and an internal or type specific one. Each type provides mapping functions to convert from one to another. The built in Tcl types fall into two broad categories depending on the mapping function they use.

A *transparent* type is one whose string representation contains all the information that the internal representation does. They have the following characteristics.

- Mapping from the string representation to the internal representation requires no other information.
- Changes to the internal representation requires changes to the string representation.
- Objects of this type have no seperate identity, an identical but seperate object can be created simply by copying the string and converting to the correct type.
- The internal representation is unique to a Tcl object.
- Objects of this type are automatically freed when they are no longer needed.

A *handle* type is one whose string representation is simply an identifier for the internal representation.

- Mapping from the string representation to the internal representation requires additional information such as a hash table.
- Changes to the internal representation does not require changes to the string representation.
- Objects of this type have their own identity, copying the string and converting to the correct type, simply results in another object which refers to the same internal representation as the original.
- The internal representation may be shared between many Tcl objects.
- Objects of this type have to be explicitly freed.

Integer, **double**, **boolean**, **string**, **regular expression** and **list** are all *transparent* types. **Command**, **index**, **window** and **font** are all *handle* types.

2.3.2 Conversions

Converting from one type to another is done as follows

1. Update the string representation.
2. Invalidate the existing internal representation.
3. Parse the string representation and create the new internal representation.
4. Change the type.

Step 2. calls the free internal representation operation of the current type which means that it is difficult to distinguish between converting from one type to another and deleting the object.

2.3.3 Shared objects

An object is shared when its reference count is greater than 1 and you have to be very careful when working with objects which are or could be shared. In fact you should assume that any object you are working with is shared, unless you know otherwise.

The one rule that you must follow when working with shared objects is that you must **never** modify the string representation: you can do just about anything else to it but that. If no string representation exists then it is valid to generate it but once it exists you must not change it. i.e.

```
set a "hello"
set b $a
append b ", world"
puts $a
```

In the above code the assignment of a to b causes the literal string object "hello" to have a reference count of 2 and hence is shared. If [append] modified the string representation of that shared object then the code would output "hello, world" which is not correct according to Tcl's semantics. Instead, if [append] has to append to a variable whose value is shared it first duplicates the value object and works with that duplicate ensuring the correct behaviour. This is called *copy-on-write* semantics.

2.3.4 Lossy conversions

A lossy conversion is one which loses information, e.g. converting from a floating point number to an integer can lose information. If an object is subject to a lossy conversion it is most important that the string representation is generated before the conversion happens.

```
set a 1
incr a
set b [expr {$a * 0.1}]
puts $a
```

In the above code a is initialised with an object which has a string representation of "1", a type of integer and a value of 1. It is then incremented, resulting in it being assigned a new object[1] with no string representation, a type of integer and a value of 2. [expr] causes the object to be converted to a double type, with a value of 2.0. If that conversion did not generate the string representation from the integer value then the final line would output 2.0, instead of 2 which is of course incorrect.

2.3.5 Literal handling

When parsing, Tcl replaces all literal strings with a Tcl object and identical literal strings are replaced with the same Tcl object. In the following code all occurrences of .b are replaced with the same Tcl object.

[1]If the object is not shared then it would be the same object.

```
button .b -command {
  puts "Button pressed"
}
pack .b
```

This behaviour has lots of advantages in that it reduces the memory requirements and if the object is used in the same context as shown above it can also improve the performance by allowing subsequent uses to benefit from any cached information. However, because this process increases the number of shared objects it also can have disadvantages because of the increased chance that shimmering (see below) will occur.

If the literal string looks like an integer then the parser will actually create an integer object, rather than simply a literal string object.

2.3.6 Shimmering

This is what happens when an object is repeatedly converted from one type to another and happens most often with shared objects but can happen with any object, because the problem is caused not by the number of references but by the number of different uses made of the object. These conversions can take up a lot of time causing performance problems. It is very difficult to actually quantify the effect shimmering has on a program as there is no way at the moment to monitor the conversions which take place. This also means that it is difficult to find these problems, unless you have a deep understanding of the internals of Tcl.

```
set b 1.0
for {set a 0} {$a < 200} {incr a 2} {
  set b [expr {$b / 2}]
}
```

In the above admittedly slightly contrived example the literal integer object 2 is used both as an integer, by [incr] and as a double, by [expr]. This means that it will be repeatedly converted from one type to another with the consequent loss of performance.

2.3.7 Threads

Accesses to Tcl objects are not serialised because the overhead would be too great and for the most part unnecessary. Because of this you must not use an object from two different threads at once. It is possible to pass objects from one thread to another but they must not be used concurrently. Also, shared objects must never be passed from one thread to another because it is almost certain that this rule would be broken as you do not have control over how and when the object is to be used.

3 Interfaces

One of the biggest limitations of Tcl objects is that they can only have one internal representation which is the primary cause of shimmering. Adding additional internal representations is not really a suitable solution for the following reasons.

- Increased memory usage in both the Tcl_Obj structure itself and for all the internal representations which now exist concurrently.
- Increased complexity managing the internal representations, choosing which one to free, finding an internal representation of the correct type, etc.
- It will most likely introduce major incompatabilities with existing extensions.

The approach that **Feather** takes is to keep one internal representation but to allow each object type to behave in different ways depending on the context, which means that in many cases conversions become unnecessary. This does not solve all shimmering problems but it does eliminate a large set of them which for the most part tend to be impossible for the Tcl programmer to do anything about.

These additional behaviours are provided by *interfaces*. If an object type can behave like a command then it provides an implementation of the command *interface*, if it can behave like a foobar then it provides an implementation of the foobar *interface*.

Conversely, if a particular context needs an object which behaves like a foobar then it checks to see if it provides a foobar *interface*. If it does then the *interface* is used, otherwise the context could try and convert the object to a known type which does provide a foobar *interface*. If the conversion failed, or no such type existed then an error would be reported.

3.1 Implementation

An *interface* definition is simply a structure of function pointers and possibly other related data. The Tcl_ObjType structure is the *interface* which Tcl object types must provide. An *interface* implementation consists of a set of functions and an initialised instance of the structure.

Rather than go into details of exactly how the *interface* mechanism works I will save space and time by just describing its important features.

- It is very dynamic.
- The Tcl_ObjType is extended to allow a set of *interfaces* to be associated with it. This is done dynamically to avoid any incompatabilities with existing extensions.
- Each *interface* definition has a unique fixed name. e.g. the **Feather** command *interface* is called "feather::command".
- Each *interface* definition used is dynamically allocated a key. This key is used to retrieve an *interface* from a Tcl_ObjType in O(1) time.
- *Interfaces* have to be added to, and removed from, Tcl_ObjTypes dynamically.

While developing the *interface* mechanism I did take a brief look at the way Python worked but rejected it immediately because it only seems to support a fixed set of *interfaces* which are hard coded into its structures. The definition of PyTypeObject in the Python header file object.h has space reserved in it for each *interface* that is supported with unused space for future expansion. This seemed far too restrictive for Tcl which prides itself on its dynamism and lack of limits.

3.2 Polymorphism

A polymorphic function, or command, is one which can work with different types of objects. *Interfaces* support polymorphism in just the same way as abstract virtual classes in C++ [1] do,

4 Opaque objects

As mentioned before Tcl only supports a few data types; strings, numbers, lists and arrays. While the latter two are very powerful they can be difficult to use and may also be relatively inefficient. **Feather** attempts to solve these problems by providing a rich set of new types.

Unfortunately, neither a *transparent* or a *handle* type was suitable for these new types. *Transparent* objects must in general be copied before being changed. This would be very inefficient for the types that I wanted to provide and make them much harder to use. On the other hand the fact that *handle* objects need to be explicitly freed makes them unsuitable because of the increased risk of memory leaks. Therefore, I created a new category of object types called *opaque*. *Opaque* objects combine parts of *transparent* and *handle* types and have the following characteristics (labelled to indicate whether they were taken from *transparent* or *handle* type)

- Mapping from the string representation to the internal representation requires additional information such as a hash table. (*Handle*)
- Changes to the internal representation does not require changes to the string representation. (*Handle*)
- Objects of this type have their own identity, copying the string and converting to the correct type, simply results in another object which refers to the same internal representation as the original. (*Handle*)
- The internal representation may be shared between many Tcl objects. (*Handle*)
- Objects of this type are automatically freed when they are no longer needed. (*Transparent*)

4.1 Mutable objects

Mutable objects consist of those *opaque* objects whose internal representation can be changed. Support for *mutable* objects was the primary reason for creating *opaque* objects although as can be seen later it is not the only one. The majority of the new **Feather** types are *mutable*.

The introduction of *mutable* objects into Tcl brings with it the ability for Tcl scripts to create cycles of Tcl objects. These cycles will not be cleared up by simple reference counting and **Feather** does not currently support garbage collection.

4.2 Implementation

The implementation of *opaque* objects is quite simple but does reveal some subtleties about their use which mean that they are not suitable for all purposes. While the following implementation is not the only way to implement *opaque*-like types I have found it to be the best both in terms of simplicity of implementation and versatility from the Tcl programmer's perspective. Also, remember that **Feather** provides a framework for creating *opaque* objects and therefore has to cope with problems in a very general manner.

The internal representations of *opaque* objects need to be reference counted because each one can be referenced by multiple Tcl objects. In fact it is the internal representation which uniquely identifies the *opaque* object, not a Tcl object.

In addition to the normal Tcl_ObjType operations, *opaque* objects also need to support the *opaque interface*. This *interface* consists of the following operations:

- Set internal representation.
- Get internal representation.
- Increment reference count.
- Decrement reference count.

The main reason why this *interface* is needed is to allow an object whose string representation refers to an *opaque* object to be converted back to that object whatever type it may be.

The following describes the Tcl_ObjType interface that *opaque* objects implement and also the *opaque interface* that they use.

4.2.1 Update the string representation

This operation has to create a string representation which uniquely identifies the internal representation amongst all *opaque* objects. At the moment **Feather** objects uses a combination of the type name, for readability, and the address of the internal representation, for simplicity.

As soon as, but not before, the string representation is generated it becomes necessary to provide a way to convert an object from its string representation to the correct type.

```
set opaque [createFooObject]
eval useFooObject [list $opaque]
```

In the above code `[eval]` concatenates the string representations of the literal string `useAsFooObject` and the string representation of the list containing the *opaque* object referred to by `opaque` and then evaluates it. At this point `[useAsFooObject]` is passed an untyped Tcl object which it needs to convert to an *opaque* **Foo** type.

While this conversion back from the string representation to the *opaque* object type is not strictly necessary, without it the objects are almost unusable because they will not work properly with `[eval]` or `[uplevel]` which are very commonly used.

Therefore, once this operation has generated the string representation it then stores the internal representation and the type of the *opaque* object into the table of *opaque* objects using the string representation as the key.

4.2.2 Convert to type

This operation uses the string representation as a key to find the internal representation and object type in the table of *opaque* objects. If it found them it then checks that the stored type and the type to which it is being converted match and only then does it do the conversion. If either the key was not valid, or the check failed then an error is generated and the conversion does not go ahead.

The conversion is actually done by the *set object from internal representation* operation from the *opaque* interface.

4.2.3 Duplicate the internal representation

If the object is *mutable* then this operation simply has to create a duplicate of the internal representation and associate it with the duplicated Tcl object, just like *transparent* objects.

If on the other hand, the object is not *mutable* then this operation can either create a duplicate of the internal representation as above, or it can just increment the reference count of the existing internal representation.

Duplicating the internal representation requires that the string representation of the duplicated object be invalidated because otherwise there would be two different *opaque* objects with the same string representation. This breaks the rule defined in the section above that the string representation has to uniquely identify the internal representation.

Invalidating the string representation of the duplicated object may seem like a dangerous thing to do but it is actually quite safe. The duplication process was designed with *copy-on-write* semantics in mind so it was never intended to create a clone of the original object but rather a copy of the object which could be modified. The string representations of duplicated objects are usually invalidated after they have been created. Because *opaque* objects do not support *copy-on-write* semantics they never need to be implicitly duplicated, rather they are explicitly duplicated when necessary.

As far as the Tcl programmer is concerned the difference between duplication and incrementing the reference count is that the former changes the string representation of the object whereas the latter does not.

4.2.4 Free the internal representation

This operation simply decrements the reference count of the internal representation. Only when the reference count reaches zero is the internal representation actually freed. Freeing the internal representation also removes the information associated with the string representation in the table of *opaque* objects.

4.2.5 Set internal representation

This operation takes a Tcl_Obj pointer and a pointer to the internal representation and converts the object to the correct type and attaches the internal representation to it.

```
set opaque [createFooObject]
eval useOpaqueObject [list $opaque]
```

In the above code `[useOpaqueObject]` is passed an untyped string object which it needs to convert to an *opaque* object, however, it does not know what the type of the object should be. The conversion process uses the string representation as a key to find the internal representation and the object type in the table of *opaque* objects. If it found them it then calls this operation to do the actual conversion.

4.2.6 Get internal representation

This operation takes a Tcl_Obj pointer and returns a pointer to the internal representation.

4.2.7 Increment reference count

This operation increments the reference count of the internal representation.

4.2.8 Decrement reference count

This operation decrements the reference count of the internal representation.

4.3 Interaction with interfaces

The process of retrieving an *interface* implementation from an object is complicated by the introduction of *opaque* objects. Without them, a single query to the object is sufficient to determine whether it supports the *interface*. With them, the object has to be converted to an *opaque* object first if necessary and then queried.

4.4 Limitations

Opaque objects have some limitations which mean that they are not suitable for all uses. n

4.4.1 Conversions

You have to be very careful when using *opaque* objects that you do not convert them to other types as this will cause the reference from the Tcl object to the internal representation to be released which may cause the internal representation to be freed.

transparent objects do not have this problem because they can be recreated from the string representation if necessary and *handle* objects have to be explicitly freed.

4.4.2 Callbacks

Opaque objects do not work well with string based callbacks.

```
set opaque [createFooObject]
scrollbar .sb -command [list command $opaque]
unset opaque
```

In the above code `[scrollbar]` takes a copy of the string representation of the list and stores it away. By the time the scrollbar command is evaluated `[unset]` has caused the *opaque* object to be freed.

The solution to this is to make sure that the *opaque* object exists as long as the `-command` option refers to it.

The following code fails in the same way, even though button commands are stored as Tcl objects, because the Tcl parser concatenates `bCommand` and the string representation of the *opaque* object before calling `[button]`.

```
set opaque [createFooObject]
button .b -command "command $opaque"
unset opaque
```

5 Container objects

A container object is simply an object which provides a container *interface*.

5.1 Interface

The container *interface* defines operations to read and write elements, to remove elements, to check that elements exist, to get the size, contents and keys of the container and also to get an iterator for the container. It can handle both single dimensional, e.g. lists, and multi-dimensional containers, e.g. matrices.

Container iterators have their own *interface* which defines operations to increment and decrement it, to read and write elements, to free it and to check that it has reached the end.

The **list** type is the default container type. This means that if an object being used as a container does not have a container *interface* and is not *opaque* then it is converted to a **list** object, which does have a container *interface* courtesy of **Feather**.

5.2 Implementation

The implementation of container objects is very simple. There is one thing however which is worth mentioning because it affects how the Tcl programmer uses them. The string representation of the *opaque* container objects contains a space to enable polymorphic functions like `[loop]` and `[container]` to differentiate between the string representation of an *opaque* container object and the string representation of a list containing a single *opaque* container object.

5.3 New container objects

Unfortunately, there is not enough space to describe all the details of each of the container types that **Feather** provides. The following list just provides a brief description of each one.

chain A *mutable* linked list which is O(1) for insertion and removal and O(N) for indexing.

hash A *mutable* hash table similar to, but faster and more efficient than, a Tcl variable array. The main reason for this is that it uses Tcl objects and not strings for the keys.

map A *mutable* container based on a red-black or 2-3-4 tree. The objects are stored in order in the tree, and searching and inserting are all very efficient. This is the perfect container to use to implement a set construct.

sequence A *transparent* container which can be used to efficiently create repetitions of objects or arithmetic series.

structure A *mutable* container very similar to C structures.

vector A *mutable* container very similar to Tcl lists.

Creation of a container object is done by calling the Tcl command of the same name with the first argument as `create`. This command also provides type specific operations. Operations which can be done through the container *interface* are all done using `[container]`.

5.4 Polymorphic container commands

5.4.1 container

This command provides access container objects through the container *interface*. The following code creates a vector and a hash object and then gets the first element in the vector (indexed by 0) and the colour of a frog.

```
set vector [vector create alpha beta gamma]
set colourOf [hash create frog green ]

container get $vector 0
container get $colourOf frog
```

5.4.2 loop

This command provides a way to iterate over containers. The following code iterates over the contents of the vector and prints them on stdout.

```
set vector [vector create alpha beta gamma]
loop letter $vector {
    puts $letter
}
```

6 Command objects

Command objects are Tcl objects which can be used in place of a normal Tcl command. The distinguishing feature of *command* objects is that they implement the *command* interface.

Although all existing **Feather** *command* objects are *opaque* they do not need to be. In fact Tk windows are a prime example of a *handle* object type just waiting to be converted into a *command* object.

6.1 Interface

The command *interface* consists of one function whose prototype is the same as a Tcl_ObjCmdProc.

cmdName is the default command type.

6.2 Implementation

Command objects are implemented by modifying the behaviour of the **cmdName** type in a similar way to Tcl Blend [2]. Unfortunately, I did not discover this until well after I had implemented by own version.

What differentiates **Feather** from Tcl Blend is that **Feather** provides a framework for others to add in their own *command* objects.

One important distinction between *opaque command* objects and other *opaque* objects is that the string representation of an *opaque command* object contains no characters which are treated specially by `[eval]`, i.e. spaces, newlines and semicolons. The reason for this is to allow them to easily pass through `[eval]` without having to worry about using `[list]` to protect them.

6.3 Lambda objects

This is basically an unnamed procedure which is wrapped up in an *opaque* object. It is *opaque* in order to hide the arguments and the body.

A lambda *command* object is created by using `[lambda]` which takes the same arguments as `[proc]` apart from the name. It should behave identically to a named procedure with the same arguments and body.

```
set command [lambda $parameters $body]
eval $command $arguments
```

The above code should behave identically to the following code ignoring any differences due to the different names.

```
proc command $parameters $body
eval command $arguments
```

Just like `[proc]`, `[lambda]` does not support static scoping, however by combining lambda objects and curried objects which are described later it is possible to emulate it.

6.3.1 Examples

The following code creates a lambda *command* object which returns the result of multiplying its argument by itself. When applied to 12 this returns 144, or 12^2.

```
set square [lambda {x} {
  expr {$x * $x}
}]
$square 12
```

The following code creates a button which, when pressed, outputs `"Pressed"`.

```
button .b -command [lambda {} {
    puts Pressed
}]
pack .b
```

6.3.2 Performance

Lambda objects are almost as fast as normal procedures and the difference between them is mainly due to the overhead arising from the overriding of the **cmdName** type. With a suitable patch to the Tcl core command objects should be just as fast as normal procedures.

6.4 Curried objects

A curried object is an *opaque command* object which encapsulates a *command*, or *command* object and a list of other Tcl objects. When the curried object is invoked it in turn invokes the encapsulated *command* with its arguments appended to the end of the encapsulated objects.

The following code creates a curried object containing a command and the literal objects 1 and 2. Invoking the curried object with the literal objects 3 and 4 results in [command] being invoked with the literal objects 1, 2, 3 and 4.

```
proc command {args} {
    puts $args
}
set object [curry command 1 2]
$object 3 4
```

As mentioned above curried objects can be used with lambda objects to allow most if not all functional programming constructs to be used. Curried objects are also very useful for callbacks and for implementing Tk window-like object systems.

6.4.1 Examples

The following code creates a procedure, compose, which takes two commands as its arguments, constructs another command object by composing those two commands together and then returns it. It then uses [compose] to create a command object which returns the result of multiplying its argument by itself and then multiplying the result of that by itself. When applied to 3 this returns 81, or $(3^2)^2$ or 3^4.

The currying is necessary because Tcl does not support static scoping and apart from substitution, which will not work properly with command objects, there is no way for the composed command to get access to the creating commands variables.

```
proc compose {f g} {
  curry [lambda {f g x} {
    $f [$g $x]
  }] $f $g
}
set quad [compose $square $square]
$quad 3
```

It is also very easy to create simple object oriented interfaces. For each instance that you want to create you first create a *mutable* object which contains the state of that instance, and then you encapsulate that along with the command which manages the state of the object in a curried object and return that as your objects command.

```
proc command {clientData method args} {
  switch -- $method {
    :
    : Modify the $clientData object here.
    :
  }
}
```

```
proc factory {args} {
  set clientData [createStateObject]
  set command [curry command $clientData]
  return $command
}
```

6.4.2 Performance

When passing large lists through [eval], or [uplevel] it is much faster to wrap them in curried objects than using [list] to protect them. This is because it eliminates the need to generate and then reparse the string representation of the list.

7 Additional features of Feather

Unfortunately, there has not been enough room to describe in detail all of the features of **Feather** so here is a brief description of some of the things which were not included.

Tables This is the unimaginative name that I have given to the very useful data type upon which the *interface* mechanism is built.

Per interpreter data This is an AssocData-like mechanism (using *tables*) which is faster O(1) and requires less memory per key than the existing mechanism based on hash tables.

Overrides This is a mechanism to change the behaviour of existing commands in much the same way as `[rename]` and `[proc]` can but it does not pollute namespaces with renamed procedures and the overrides can be removed in any order.

Handles These are general purpose handles which can be used to safely pass Tcl objects through hazardous environments such as string based callbacks.

Generic interface This allows a Tcl object type to choose what interfaces each object of that type provides.

Generic object The generic object uses this feature to expose interfaces to the Tcl programmer.

8 The future of Feather

Feather is nowhere near finished yet, here is a selection of the things which need doing to it and with it.

- Work with Scriptics to integrate some of the **Feather** stuff into the Tcl core.
- Finish off the existing container types, and create new ones such as matrices.
- Define new *interfaces* such as a numeric one for use by `[expr]`.
- Serialisation of objects in both binary and ascii format.
- Create yet more object oriented systems using the **Feather** concepts.
- Try and define a mechanism to allow one object to be wrapped around another.
- Garbage collector to clean up reference loops created by *mutable* objects.
- Create some new types to provide efficient communication between threaded interpreters.

9 Acknowledgments

Thanks to everyone who helped with the design and implementation of **Feather** and also to those who helped with this paper.

A special thank you to Jean-Claude Wippler for yet another good idea and to Alexandre Ferrieux for spending the time to convince me that conversion from string to opaque object was needed.

The biggest thank you must however be reserved for my wife Julie who read about 4 or 5 different versions of this paper even though she did not under it.

10 Availability

At the time of writing I have not yet managed to get permission to make the source code freely available. As soon as I get it I will announce it on comp.lang.tcl and comp.lang.tcl.announce.

References

[1] Bjorn Stroustrup, *The C++ Programming Language, 3rd Edition*, Addison-Wesley Publishers (1997).

[2] Tcl Blend, http://www.scriptics.com/products/java/, http://ptolemy.eecs.berkeley.edu/ cxh/java/

[3] comp.lang.tcl

The reference section is very short because I have not had access to conference papers before and most of my knowledge about what is happening in the rest of the Tcl world is obtained from [3].

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- On-line access to papers from the USENIX Conferences and Symposia from the day the meeting starts, and on-line access to.issues of *;login:* shortly after publication.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal, The Perl Journal, IEEE Concurrency, Server/Workstation Expert*, *Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C/C++ Users Journal
Cisco Systems, Inc.
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
UUNET Technologies, Inc.
Web Publishing, Inc.
Windows NT Systems Magazine

Sage Supporting Members:

Collective Technologies
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
TransQuest Technologies, Inc.
Unix Guru Universe

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org*.
URL: *http://www.usenix.org*.

ISBN 1-880446-24-3